USAISEC

US Army Information Systems Engineering Command
Fort Huachuca, AZ 85613-5300

U.S. ARMY INSTITUTE FOR RESEARCH
IN MANAGEMENT INFORMATION,
COMMUNICATIONS, AND COMPUTER SCIENCES

SAMeDL:
Technical Report Appendix D –
Language Reference Manual

ASQB-GI-92-017

September 1992

AIRMICS
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188
Exp. Date: Jun 30, 1986

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT N/A
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) N/A
6a. NAME OF PERFORMING ORGANIZATION / 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION N/A
6c. ADDRESS (City, State, and Zip Code)	7b. ADDRESS *(City, State, and ZIP Code)* N/A
8b. NAME OF FUNDING/SPONSORING ORGANIZATION Software Technology Branch, ARL / 8b. OFFICE SYMBOL *(If applicable)* AMSRL-CI-CD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 115 O'Keefe Bldg. Georgia Institute of Technology Atlanta, GA 30332-0800	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
SAMeDL: Technical Report Appendix D - Language Reference Manual

12. PERSONAL AUTHOR(S)
MS. Deb Waterman

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical Paper	FROM Apr 91 TO Sept 92	Sept 15, 1992	125

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUBGROUP	Ada Database Access, SAMeDL, Ada extension module, SQL

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report details the research efforts into the SQL Ada Module Database Description Language (SAMeDL). Four compilers are presented (Oracle, Informix, XDB, and Sybase) that allow Ada application programs to access database using a standard SQL query language. Copies of the compiler can be obtained from the DoD Ada Joint Program Office 703/614-0209.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)*	22c. OFFICE SYMBOL
LTC David S. Stevens	(404) 894-3110	AMSRL-CI-CD

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

This research was performed by Statistica Inc., contract number DAKF11-91-C-0035, for the Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), the RDTE organization of the U. S. Army Information Systems Engineering Command (USAISEC). This final report discusses a set of SAMeDL compilers and work enviornment that were developed during the contract. Request for copies of the compiler can be obtained from the DoD Ada Joint Program Office, 703/614/0209. This research report is not to construed as an official Army or DoD Position, unless so designated by other authorized documents. Material included herein is approved for public release, distribution unlimited. Not protected by copyright laws.

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED

Glenn E. Racine, Chief
Computer and Information
Systems Division

James D. Gantt, Ph.D.
Director
AIRMICS

APPENDIX D

SAMeDL Language Reference Manual

SAMeDL Language Reference Manual

Intermetrics, Inc.

Document IR-VA-011-1
Date 07-July-1992

Published by
Intermetrics, Inc.
733 Concord Avenue, Cambridge, Massachusetts 02138

Table of Contents

Chapter 1 Introduction

1.1 Scope

This manual defines the **SQL Ada Module Extensions Description Language** (**SAMeDL**). The language described herein is strongly based on the draft language outlined by Marc Graham in [SAME].

The description in this manual assumes an underlying working knowledge, on the part of the reader, of the SAME methodology [**SAMEGuide**], the SQL standard [**SQL**], and the Ada standard [**Ada**].

1.2 Notation

The notation used in this manual to specify language constructs is based on the **Backus-Naur Form** (**BNF**), which uses grammar rules to specify the syntax of a language. The syntax of a language defines what sequences of symbols are legal in that language.

A BNF grammar consists of a set of *terminal symbols*, a set of *non-terminal symbols*, and a set of *productions* (or rewrite rules).

Non-terminal symbols are expanded by rewrite rules. They represent program constructs such as statements and expressions.

Terminal symbols are not expanded by rewrite rules. They represent program symbols such as reserved words and punctuation marks.

A production is a rewrite rule that allows a non-terminal symbol to be replaced by a (possible empty) sequence of terminal and non-terminal symbols.

The following naming conventions are used within the grammar rules:

- Lower case names (abstract_module, constant_declaration, etc.) represent non-terminal symbols.
- Lower case names that are bold-faced (**abstract**, **record**, etc.) and bold-faced strings (=, <>, etc.) represent terminal symbols.
- The italicized prefixes *Ada* and *SQL*, when appearing in the names of syntactic categories, indicate that an Ada or SQL syntactic category has been incorporated into this document. For example, the category *Ada*_identifier is identical to the category identifier as described in section 2.3 of [**Ada**]; whereas the category *SQL*_identifier is identical to the category identifier as described in section 5.3 of [**SQL**].
- Numerical suffixes attached to the names of syntactic categories are used to distinguish appearances of the category within a rule or set of rules.

The rules of productions are applied as follows:

1. There is a special non-terminal symbol, called the *start symbol*, from which all legal sequences are generated. For example, the start symbol for the SAMeDL grammar is *compilation_unit*.

2. Each production is of the form

   <non-terminal symbol> ::= <sequence of symbols>

   and is interpreted as "the non-terminal on the left hand side may be replaced by the sequence of symbols on the right hand side." For example

   a ::= b c

   means that "a" may be replaced by "b c".

3. The symbol '|' may be used on the right hand side of a production to indicate a choice of replacements. For example

   a ::= b | c

   means that "a" may be replaced by either "b" or "c".

4. The symbols '[' and ']' signify that the enclosed sequence is optional. For example

   a ::= b [c]

   means that "c" is optional, and therefore "a" may be replaced by either "b" or "b c".

5. The symbols '{' and '}' signify the repetition (possibly 0 times) of the enclosed sequence. For example

   a ::= b { b }

   means that "a" may be replaced by one or more "b" symbols.

1.3 Forward References

In order that a given section give thorough coverage of its subject, it is often necessary to employ terms, or to refer to grammatical productions, which have not yet appeared in the text. Generally references to the appropriate chapter or section will appear. For convenience, an alphabetic summary of the entire grammar of the language appears in Appendix F.

1.4 References

1. **[Ada]** *Reference Manual for the Ada Programming Language*, Ada Joint Program Office, 1983.

2. **[ESQL]** *Database Language - Embedded SQL*, American National Standards Institute, X3.168-1989, 1989.

3. **[SAME]** *The SQL Ada Module Description Language, Intermediate Version 3*, Software Engineering Institute/Carnegie Mellon University, 21 November 1991.

4. **[SAMEGuide]** *Guidelines for the Use of the SAME*, Marc H. Graham: Software Engineering Institute/Carnegie Mellon University, Technical Report CMU/SEI-89-TR-16, May 1989.

5. **[SQL]** *Database Language - SQL*, American National Standards Institute, X3.135-1989, 1989.

6. **[User]** *SAMeDL Development Environment User Manual*, Intermetrics, Inc., IR-VA-012, 28 February 1992.

1.5 Design Goals And Language Summary

1.5.1 Design Goals

The SQL Ada Module Description Language (SAMeDL) is a Database Programming Language designed to automate the construction of software conforming to the SQL Ada Module Extensions (SAME) application architecture (see **[SAMEGuide]**).

The SAME is a *modular* architecture. It uses the concept of a Module as defined in **[SQL]** and **[ESQL]**. As a consequence, a SAME-conforming Ada application does not contain embedded SQL statements and is not an embedded SQL Ada program as defined in **[ESQL]**. Such a SAME-conforming application treats SQL in the manner in which Ada treats all other languages: it imports complete functional modules, not language fragments.

Modular architectures treat the interaction of the application program and the database as a design object. This results in a further isolation of the application program from details of the database design and implementation and improves the potential for increased specialization of software development staff.

Ada and SQL are vastly different languages: Ada is a Programming Language designed to express algorithms, which SQL is a Database Language designed to describe desired results. Text containing both Ada and SQL is therefore confusing and difficult to maintain. SAMeDL is a Database Programming Language designed to support the goals and exploit the capabilities of Ada with a language whose syntax and semantics is based firmly in SQL. Beyond modularity, the SAMeDL provides the application programmer the following services:

- An abstract treatment of null values. Using Ada typing facilities, a safe treatment of missing information based on SQL is introduced into Ada database programming. The treatment is safe in that it prevents an application from mistaking missing information (null values) for present information (non-null values).

- Robust status code processing. SAMeDL's Standard Post Processing provides a structured mechanism for the processing of SQL status parameters.

- Strong typing. SAMeDL's typing rules are based on the strong typing of Ada, not the permissive typing of SQL.

- Extensibility. The SAMeDL supports a class of user extensions. Further, it controls, but does not restrict, implementation defined extensions.

1.5.2 Language Summary

1.5.2.1 Overview

The SAMeDL is designed to facilitate the construction of Ada database applications that conform to the SAME architecture as described in **[SAMEGuide]**. The SAME method involves the use of an abstract interface, an abstract module, a concrete interface, and a concrete module.

The abstract interface is a set of Ada package specifications containing the type and procedure declarations to be used by the Ada application program. The abstract module is a set of bodies for the abstract interface. These bodies are responsible for invoking the routines of the concrete interface, and converting between the Ada and the SQL data and error representations. The concrete interface is a set of Ada specifications that defined the SQL procedures needed by the abstract module. The concrete module is a set of SQL procedures that implement the concrete interface.

Within this document, the concrete module of **[SAMEGuide]** is called an SQL module and its contents are given under the headings **SQL Semantics**. The abstract modules of **[SAMEGuide]** are given under the heading **Ada Semantics**.

1.5.2.2 Compilation Units

A *compilation unit* consists of one or more modules. A module may be either a definitional module containing shared definitions, a schema module containing table, view, and privilege definitions, or an abstract module containing local definitions and procedure and cursor declarations.

1.5.2.3 Modules

A *definitional module* contains the definitions of base domains, domains, constants, records, enumerations, exceptions, and status maps. Definitions in definitional modules may be seen by other modules.

A *schema module* contains the definitions of tables, views, and privileges.

An *abstract module* defines (a portion of) an application's interface to the database: it defines SQL services needed by an Ada application program. An abstract module may contain procedure declarations, cursor declarations, and definitions such as those that may appear in a definitional module. Definitions in an abstract module, however, may not be seen by other modules.

1.5.2.4 Procedures and Cursors

A *procedure* declaration defines a basic database operation. The declaration defines an Ada procedure declaration and a corresponding SQL procedure. A SAMeDL procedure consists of a single statement along with an option input parameter list and an optional status clause. The input parameter list provides the mechanism for passing information to the database at runtime. A statement in a SAMeDL procedure may be a commit statement, rollback statement, insert statement query, insert statement values, update statement, select statement or an implementation-defined extended statement. The semantics of a SAMeDL statement directly parallel that of its corresponding SQL statement.

SAMeDL *cursor* declarations directly parallel SQL cursor declarations. In contrast to the language in **[SQL]**, the procedures that operate on cursors, procedures containing either an open, fetch, close, update positioned or delete positioned statement, are packaged with the declaration of the cursor upon which they operate, thereby improving readability. Further, if no procedure containing an open, fetch or close statement is explicitly given in a cursor declaration, the language provides such procedures implicitly, thereby improving ease of use.

1.5.2.5 Domain and Base Domain Declarations

Objects in the language have an associated *domain*, which characterizes the set of values and applicable operations for that object. In this sense, a domain is similar to an Ada type.

A *base domain* is a template for defining domains. A base domain declaration consists of a set of parameters, a set of patterns and a set of options. The parameters are used to supply information needed to declare a domain or subdomain derived from the base domain. Patterns contain templates for the generation of Ada code to support the domain in Ada applications. This code generally contains type declarations and package instantiations. Options contain information needed by the compiler. Parameters may be used in the patterns and options and their values may be referenced in other statements.

Base domains are classified according to their associated data class. A data class is either integer, fixed float, enumeration, or character. A numeric base domain has a data class of enumeration, and defines both an ordered set of distinct enumeration literals and a bijection between the enumeration literals and their associated database values. A character base domain has a data class of character.

1.5.2.6 Other Declarations

Certain SAMeDL declarations are provided as a convenience for the user. For example, *constant* declarations name and associate a domain with a static expression. *Record* declarations allow distinct procedures to share types. An *exception* declaration defines an Ada exception declaration with the same name.

1.5.2.7 Value Expressions and Typing

Value expressions are formed and evaluated according to the rules of SQL, with the exception that the strong typing rules are based on those of Ada. In the typing rules of the SAMeDL, the domain acts as an Ada type in a system without user defined operations. Strong typing necessitates the introduction of domain conversions. These conversions are modeled after Ada type conversions; the operational semantics of the SAMeDL domain conversion is the null operation or identity mapping. The language rules specify that an informational message be displayed under circumstances in which this departure from the Ada model has visible effect.

1.5.2.8 Standard Post Processing

Standard post processing is performed after the execution of an SQL procedure but before control is returned to the calling application procedure. The *status clause* from a SAMeDL procedure declaration attaches a *status mapping* to the application procedure. That status mapping is used to process SQL status data in a uniform way for all procedures and to present SQL status codes to the application in an application-defined manner, either as a value of an enumerated type, or as a user defined exception. SQL status codes not specified by the status map result in a call to a standard database error processing procedure and the raising of the predefined SAMeDL exception, SQL_Database_Error. This prevents a database error from being ignored by the application.

1.5.2.9 Extensions

The data semantics of the SAMeDL may be extended without modification to the language by the addition of user-defined base domains. For example, a user-defined base domain of DATE may be included without modification to the SAMeDL.

DBMS specific (i.e., non-standard) operations and features that require compiler modifications (e.g., dynamic SQL) may also be included into the SAMeDL. Such additions to the SAMeDL are referred to as extensions. Schema elements, table elements, statements, query expressions, query specifications, and cursor statements may be extended. The modules, tables, views, cursors, and procedures that contain these extensions are marked (with the keyword **extended**) to indicate that they go outside the standard.

1.5.2.10 Default Values in Grammar

Obvious but over-ridable defaults are provided in the grammar. For example, open, close, and fetch statements are essential for a cursor, but their form may be deduced from the cursor declaration. The SAMeDL will therefore supply the needed open, close, and fetch procedure declarations if they are not supplied by the user.

Chapter 2 Lexical Elements

SAMeDL compilation units are sequences of lexical elements, which represent operators, delimiters, reserved words, identifiers, and numbers. These lexical elements correspond to the terminal symbols that appear in the grammar rules that define the syntax of SAMeDL.

2.1 Character Set

The only characters allowed in the text of a compilation are the basic characters and the characters that make up character literals (described in Section 2.4). Each character in the basic character set is represented by a graphical symbol.

```
basic_character ::=  letter              |
                     digit               |
                     special_character   |
                     space_character
```

The characters included in each of the above categories or the basic characters are defined as follows:

1. **letter**
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
   a b c d e f g h i j k l m n o p q r s t u v w x y z

2. **digit**
   0 1 2 3 4 5 6 7 8 9

3. **special_character**
   ' ( ) * + , - . / : ; < = > |

4. **space_character**

2.2 Lexical Elements, Separators, And Delimiters

The text of a SAMeDL compilation unit is a sequence of *lexical elements*. Each lexical element is either a delimiter, an identifier (which may be a reserved word), a literal, or a comment.

Blanks, tabs, newlines, and comments are considered to be *separators* provided they do not appear within other lexical elements (i.e., a comment or a literal). One or more separators are allowed between adjacent lexical elements; explicit separators are required between adjacent lexical elements when they could be interpreted as a single lexical element without separation.

A *delimiter* is either one of the following special characters

```
( ) * + , - . / : ; < = > |
```

or one of the following *compound delimiters* each composed of two adjacent special characters

```
=> .. := <> >= <=
```

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment or literal.

The remaining lexical element forms are described in the following sections of this chapter.

2.3 Identifiers

Identifiers are used as names and also as reserved words. In general, they take the form of Ada identifiers (see **[Ada]** Section 2.3). The exception is the use of *SQL*_identifiers as the names of schemas, tables, views, and columns. The major difference between *SQL*_identifiers and *Ada*_identifiers is that *SQL*_identifiers are limited to 18 characters in length, whereas *Ada*_identifiers are essentially unlimited in length. An SQL reserved word shall not appear at a point where the grammar specifies an *SQL*_identifier, and an Ada reserved word shall not appear at a point where the grammar specifies an *Ada*_identifier.

```
identifier ::= letter { [ underline ] letter_or_digit }

underline ::= _

letter_or_digit ::= letter | digit
```

The function SQL_{NAME} that appears in this language is an approximation of an injection (one-to-one function) from the set of *Ada*_identifiers to the set of *SQL*_identifiers, in the sense that, if I_1 and I_2 are distinct *Ada*_identifiers within a SAMeDL compilation unit, then $SQL_{NAME}(I_1)$ is distinct from $SQL_{NAME}(I_2)$.

2.4 Literals And Data Classes

SAMeDL literals follow the SQL literal syntax and are categorized into five data type classes: *character, integer, fixed, float,* and *enumeration.*

```
literal ::=  database_literal | enumeration_literal

database_literal ::=   character_literal    |
                       integer_literal      |
                       fixed_literal        |
                       float_literal

character_literal ::= ' { character } '

character ::= implementation defined

integer_literal ::= digit { digit }

fixed_literal ::=    integer_literal . integer_literal   |
                     . integer_literal                   |
                     integer_literal .

float_literal ::= fixed_literal exp [ + | - ] integer_literal

exp ::= e | E

enumeration_literal ::= Ada_identifier
```

1. Each literal L has an associated data class, denoted throughout this manual as DATACLASS(L). In particular, if L is a character, integer, fixed, float, or enumeration literal, the associated data class for L is **character**, **integer**, **fixed**, **float**, or **enumeration** respectively.

2. Every character_literal *CL* has an associated length LENGTH(CL) in the sense of [SQL], section 5.2, rule 2. For any non-character literal, *L*, LENGTH(L) = NO_LENGTH.

3. Integer, fixed, and float literals are collectively known as numeric literals; furthermore, integer and fixed literals are known as exact numeric literals while float literals are known as approximate numeric literals. Every numeric literal NL has a scale, SCALE(NL). An integer literal has scale 0. The scale of a fixed literal is the number of digits appearing to the right of the decimal point within the literal. The scale of a float literal is equal to the scale of any other float literal and is larger than the scale of any non-float numeric literal. Any non-numeric literal *L*, has SCALE(L) = NO_SCALE. See section 3.4 for the interpretation of enumeration_literals.

4. The single quote or "tic" character can be included in a character_literal by duplicating the tic. For example, the string '**tic** '''' represents a character string literal of length 5 containing the characters: t, i, c, space, and tic.

Examples:

```
''                          -- the null character string

''''                        -- character string of length 1 containing "tic"

'a character string'        -- character string of length 18

012   12                    -- integer literals having the value 12

0.5   .5   1.               -- fixed literals

1.0E-5   .5e10   .5E+8      -- float literals
```

2.5 Comments

Comments are used to document the program for purposes of readability and maintainability. They do not affect the meaning of the program, and are present solely for the enlightenment of the human reader.

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a SAMeDL compilation unit.

2.6 Reserved Words

The identifiers listed below are *reserved words* and are reserved for special significance in the language. For readability of this manual, reserved words will appear "**boldfaced**".

abstract
all
and
any
as
asc
authorization
avg

base
between
body
boolean
by

check
class
close
commit
connect
constant
conversion
count
current
cursor

data
dblength
dbms
declare
default
definition
delete
derived
desc
distinct
domain

end
enumeration
escape
exception
exists
extended

fetch
for
foreign
from
function

grant
group

having

image
in
insert
into
is

key

length
like

map
mark
max
min
module

name
named
new
not
null

of
on
open
option
or
order
out

pattern
pos
primary
privileges
procedure
public

raise
record
references
rollback

scale
schema
select
set
some
status
subdomain
sum

table
to
type

union
unique
update
use
user
uses

values
view

where
with
work

Chapter 3 Common Elements

3.1 Compilation Units

A *compilation unit* is the smallest syntactic object that can be successfully processed by the SAMeDL compiler. It consists of a sequence of one or more modules.

compilation_unit ::= module { module }

module ::= definitional_module | schema_module | abstract_module

3.2 Context Clauses

The context clauseis the means by which a SAMeDL module gains visibility to names defined in other modules. The syntax and semantics of SAMeDL context clauses are similar to the syntax and semantics of Ada context clauses (see **[Ada]** 8.4, 10.1.1).

context ::= context_clause { context_clause }

context_clause ::= with_clause | with_schema_clause | use_clause

with_clause ::= **with** module_name [as_phrase]
{ , module_name [as_phrase] } ;

use_clause ::= **use** module_name { , module_name } ;

with_schema_clause ::= **with schema** schema_name [as_phrase]
{ , schema_name [as_phrase] } ;

module_name ::= *Ada*_identifier

schema_name ::= *SQL*_identifier

as_phrase ::= **as** *Ada*_identifier

1. Consider the following with_clause and with_schema_clause:

   with M [**as** N_1];
   with schema S [**as** N_2];

   In these clauses, M shall be the name of a definitional module and S the name of a schema module. The name M of the definitional module is said to be *exposed* if the as_phrase is not present in the context_clause; otherwise the name M is *hidden* and the name N_1 is the exposed name of M. Similar comments apply to S and N_2. The name of a module (see Sections 4.1, 4.2, and 5.1) is its exposed name within the text of that module. Within the text of any module, no two exposed module names shall be the same.

2. A module_name in a use_clause shall be the exposed name of a definitional_module that is an operand of a prior with_clause.

3. The scope of a with_clause or use_clause in the context of a module is the text of that module.

4. Only an abstract or schema module context may contain a with_schema_clause.

Note: As a consequence of these definitions, abstract modules cannot be brought into the context of (**with**ed by) another module.

3.3 Table Names and the From Clause

The table names in insert, update, and delete statements and the from clauses of select statements, cursor declarations, and subqueries (see Sections 5.3, 5.4, and 8.12) also make names, in particular column names, visible. The from_clause differs from an SQL_from_clause ([SQL] 5.20) only in the optional appearance of the **as** keyword, which is inserted for uniformity with the remainder of the language.

from_clause ::= **from** table_ref { , table_ref }

table_ref ::= table_name [[**as**] correlation_name]

table_name ::= [schema_ref .] *SQL*_identifier

schema_ref ::= schema_name | *Ada*_identifier

correlation_name ::= *SQL*_identifier

1. If present, schema_ref shall be either the schema_name in the authorization clause of the abstract module in which the table_name appears (see Section 5.1) or the exposed name of a schema module in the context of the module in which table_name appears (see Section 3.2). In either case, the *SQL*_identifier shall be the name of a table within that schema module. If the schema_ref is absent from the table_name, then the *SQL*_identifier shall be the name of a table within the schema module named in the authorization clause of the module in which the table name appears.

2. If the correlation name is not present in a table_ref, then the table name in the table_ref is *exposed*; otherwise the table_name is hidden and the correlation_name is exposed. No two exposed names within a from_clause shall be the same.

3. For the scope of table_names see [SQL] section 5.20, syntax rule 4; section 8.5, syntax rule 3; and section 8.12, syntax rule 5.

3.4 References

The rules concerning the meaning of references are modeled on those of Ada and those of SQL. As neither module nesting nor program name overloading occurs, these rules are fairly simple, and are therefore listed. For the purposes of this clause, an *item* is either:

- A definitional, schema, or abstract module (Sections 4.1, 4.2, 5.1)
- A procedure, a cursor, or a procedure within a cursor (Sections 5.2, 5.4, 5.5)
- Anything in the syntactic category *definition* (Section 4.1)

- A domain parameter (Section 4.1.1.1)
- An enumeration literal within an enumeration (Sections 2.4, 4.1.6)
- An exception (Section 4.1.7)
- An input parameter of a procedure or cursor declaration (Sections 5.2, 5.4, and 5.6)
- A table defined within a schema module (Section 4.2)
- A column defined within a table (Section 4.2)

A location within the text of a module is said to be a *defining* location if it is the place of:

- The name of an item with the item's declaration (***Note:*** this includes enumeration literals within the declaration of an enumeration and domain parameters within the declaration of a domain)
- The name of a table in a from_clause
- The name of the target table of an insert, update, or delete statement
- A schema_name or module_name in a context_clause

Text locations not within comments that are not defining locations are *reference* locations. An identifier that appears at a reference location is a reference to an item. The meaning of that reference in that location, that is, the identity of the item referenced, is defined by the rules of this clause. When these rules determine more than one meaning for an identifier, then all items referenced shall be enumeration literals.

module_reference ::= *Ada*_identifier

schema_reference ::= schema_name | *Ada*_identifier

base_domain_reference ::= [module_reference .] *Ada*_identifier

domain_reference ::= [module_reference .] *Ada*_identifier

domain_parameter_reference ::= domain_reference.*Ada*_identifier

subdomain_reference ::= [module_reference .] *Ada*_identifier

enumeration_reference ::= [module_reference .] *Ada*_identifier

enumeration_literal_reference ::= [module_reference .] *Ada*_identifier

exception_reference ::= [module_reference .] *Ada*_identifier

constant_reference ::= [module_reference .] *Ada*_identifier

record_reference ::= [module_reference .] *Ada*_identifier

procedure_reference ::= [module_reference .] *Ada*_identifier

cursor_reference ::= [module_reference .] *Ada*_identifier

cursor_proc_reference ::= [cursor_reference .] *Ada_identifier*

input_reference ::= [procedure_reference .] *Ada_identifier* |
[cursor_proc_reference .] *Ada_identifier*

status_reference ::= [module_reference .] *Ada_identifier*

table_reference ::= [schema_reference .] *SQL_identifier*

column_name::= *SQL_identifier*

column_reference ::= [table_reference .] column_name

A reference is a simple name (an identifier) optionally preceded by a *prefix* : a sequence of as many as three identifiers, separated by dots. Unlike Ada (see **[Ada]** sections 8.2, 8.3), it is necessary to treat the prefix as a whole, not component by component.

For the purposes of this clause, the *text of a cursor* does not include the text of the procedures, if any, contained within the cursor. A dereferencing rule is said to *determine a denotation* for a reference if it either (i) specifies an item to which the reference refers, or (ii) determines that the reference is not valid.

Prefix Denotations

The prefix of a reference shall denote one of the following:

- An abstract module, procedure, cursor or cursor procedure, but only from within the text of the abstract module, procedure, cursor, or cursor procedure.
- A table, if the table is in scope at the location in which the reference appears.
- A domain.
- A definitional or schema module.

Note: As a consequence of the rule given earlier, that all meanings of an identifier with multiple meanings must be enumeration literals, a prefix may have at most one denotation or meaning, as it may not denote an enumeration literal.

Let *L* be the reference location of prefix *P*. Let *X*, *Y*, and *Z* be simple names. Then:

1. If *L* is within the text of a cursor procedure *U*, then *P* denotes
    a. The cursor procedure *U* if either
        i. *P* is of the form *X* and *X* is the simple name of *U;* or
        ii. *P* is of the form *X.Y; X* is the name of the cursor containing *L* (and therefore also *U*); in which case *Y* shall be the simple name of *U* else the prefix is not valid;
        iii. *P* is of the form *X.Y.Z; X* is the name of the module containing *L*; *Y* is the name of the cursor continuing *L* (and therefore also *U*); in which case *Z* shall be the simple name of *U* else the prefix is not valid;

b. The table *T* being updated in a cursor_update_statement, if the statement within the cursor procedure containing *L* is a cursor_update_statement and either

i. *P* is of the form *X* and *X* is the simple name of *T;* or

ii. *P* is of the form *X.Y; X* is an exposed name for the schema module *S* containing the declaration of *T* and *Y* is the simple name of *T*.

2. If rule 1 does not determine a denotation for *P*, then *P* denotes

a. The cursor or procedure *R*, if *L* is within the text of *R* and either

i. *P* is of the form *X* and *X* is the simple name of *R;* or

ii. *P* is of the form *X.Y; X* is the name of the module containing *L* (and therefore also *R*); *Y* is the simple name of *R* ;

b. The table *T*, if *L* is in the scope of table name *T* (see Section 3.3) and either

i. *P* is of the form *X* and *X* is the simple name exposed for *T;* or

ii. *P* is of the form *X.Y; X* is the exposed name of the schema module containing the table *T* and *Y* is a simple name of *T*.

3. If rules 1 and 2 do not determine a denotation for *P*, then *P* denotes the domain *R*,

a. If *P* is of the form *X* and *X* is the simple name of *R* and the declaration of *R* appears in the module containing *L* and precedes *L* within that module; or

b. *P* is of the form *X.Y*; *X* is the exposed name of the module containing the declaration of *R* and *Y* is the simple name of *R*.

4. If none of the above rules determines a denotation for *P*, then *P* is a simple name that denotes the

a. Definitional module *M* if either

i. *L* is in the scope of a with_clause exposing *P* as the name of *M*; or

ii. *L* is in the definitional module *M* and *P* is the name of *M*.

b. Schema module *S* if either

i. *L* is in the scope of a with_schema_clause exposing *P* as the name of *S*; or

ii. *L* is in an abstract module whose authorization clause identifies *S* and *P* is the name of *S*;

c. Abstract module *M* if *L* is within the text of *M* and *P* is the name of *M*;

d. Domain *D*, if *D* is declared within a module *N* such that there is a use clause for *N* in the module containing *L*, and *P* is the name of *D*.

Denotations of Full Names

Let *L* be the location of a reference *ld*. Then *ld* is a reference to the item *lm* if *lm* is not a module, procedure, cursor or cursor procedure, or table and

1. *ld* is of the form *P.X* where *X* is the name of *lm* and *P* is a prefix denoting
    a. A definitional module containing the declaration of *lm;*
    b. The abstract module, *M*, in which *L* appears, and *lm* is declared in *M* at a text location that precedes *L;*
    c. The procedure, cursor, or cursor procedure that contains *L*, and *lm* is an input parameter to that procedure, cursor, or cursor procedure;
    d. A table, in which case *lm* is a column within that table;
    e. A schema module, in which case *lm* is a table within that module.
    f. A domain, in which case *lm* is a parameter in that domain.
2. *ld* is of the form *X* and *X* is the name of *lm*. Then
    a. *L* appears in a cursor, procedure, or cursor procedure and
        i. *lm* is an input parameter to that cursor, procedure, or cursor procedure;
        ii. *lm* is a column of one of the tables in scope of *L*;
    b. If rule (a) does not determine a denotation for *ld*, then *lm* is declared in the module containing *L* at a location preceding *L*;
    c. If neither rule (a) nor (b) determines a denotation for *ld*, then *lm* is declared within a module *M* such that the module containing *L* has a use clause for *M*.

Note: An item *lm* is *visible* at location *L* if there exists a name *ld* (either simple or preceded by a prefix) such that if *ld* were at location *L*, then *ld* would be a reference to *lm*.

3.5 Assignment Contexts and Expression Conformance

A value expression (see Section 5.10) is said to appear in an *assignment context* if it is either:

1. The static expression in a constant declaration (see Section 4.1.2);
2. A select parameter (see Section 5.7);
3. A value in an insert_value_list (see Section 5.8); or
4. The right hand side of a set_item within an update_statement (see Section 5.3).

A value expression *VE* is said to *conform* to a domain *D* under the following conditions:

1. If DOMAIN(*VE*) ≠ NO_DOMAIN, then DOMAIN(*VE*) = *D*.

2. If DATACLASS(*D*) is integer or fixed, then DATACLASS(*VE*) is integer or fixed.

3. If DATACLASS(*D*) is float, then DATACLASS(*VE*) is integer, fixed, or float.

4. If DATACLASS(*D*) is character, then DATACLASS(*VE*) is character.

5. If DATACLASS(*D*) is enumeration, then if DOMAIN(*VE*) = NO_DOMAIN, then *VE* is an enumeration literal in *D*.

3.6 Standard Post Processing

Standard post processing is the processing that is done after execution of an SQL procedure, but before control is returned to the calling application. That processing is described as follows:

1. If a status map is attached to the procedure (see Section 5.13), then if that map contains an sqlcode_assignment whose left-hand side is equal to the value of the SQLCODE parameter, then the Ada procedure's status parameter is set to the value of the right hand side of the sqlcode_assignment, if that right hand side is an *Ada*_enumeration_literal; if that right hand side is a **raise** statement, then the named *Ada*_exception is raised. This is not considered an error condition in the sense of the next paragraph. In particular, SQL_Database_Error_Pkg.Process_Database_Error is not called.

2. If the value of the SQLCODE parameter is not matched by the left hand side of any sqlcode_assignment in the map attached to the procedure *or* there is no status map attached to the procedure and the value of the SQLCODE parameter is other than zero, then an error condition exists. In this case the parameterless procedure SQL_Database_Error_Pkg.Process_Database_Error is called. The exception SAMeDL_Standard.SQL_Database_Error is raised.

3.7 Extensions

Extended tables, views, modules, procedures, and cursors allow for the inclusion into the SAMeDL of DBMS-specific, that is, non-standard, operations and features, while preserving the benefits of standardization. These DBMS-specific extensions may be *verbs*, such as connect and disconnect, that signal the beginning and end of program execution, or *functions*, such as date manipulation routines, that extract the month from a date. The use of extensions, particularly the **extended** keyword, serves to mark those modules, tables, views, cursors, and procedures that go outside the standard and may require effort should the underlying DBMS be changed.

extended_schema_element ::= *implementation defined*

extended_table_element ::= *implementation defined*

extended_statement ::= *implementation defined*

extended_query_expression ::= *implementation defined*

extended_query_specification ::= *implementation defined*

extended_cursor_statement ::= *implementation defined*

Details concerning implementation defined extended features can be found in **[User]**.

Chapter 4 Data Description Language and Data Semantics

4.1 Definitional Modules

Definitional modules contain declarations of one or more declarations of elements such as domains, constants, records, enumeration types, and status maps. An Ada library unit package declaration is defined for each definitional module.

```
definitional_module ::=   [ context ]
                          [ extended ]
                          definition module Ada_identifier_1 is
                              { definition }
                          end [ Ada_identifier_2 ] ;

definition ::=  base_domain_declaration   |
                domain_declaration        |
                subdomain_declaration     |
                constant_declaration      |
                record_declaration        |
                enumeration_declaration   |
                exception_declaration     |
                status_map_declaration
```

When present, *Ada*_identifier_2 shall equal *Ada*_identifier_1.

Notes:

- No with_schema_clause shall appear in the context of a definitional module (see Section 3.2).
- No two declarations within a definitional module shall have the same name, except for enumeration literals (see Section 4.1.6).

Ada Semantics

For each definitional module within a compilation unit there is a corresponding Ada library unit package generated which has the same name as the definitional module. For each definition within the definitional module, an Ada construct which provides the appropriate Ada semantics for the definition will be generated and placed within the specification of that package.

4.1.1 Base Domain Declarations

Base domains are the basis on which domains are defined. A base domain declaration has three parts: a sequence of parameters, used in domain declarations to supply information to the other two parts; a sequence of patterns, used to produce Ada source code in support of a domain; and a sequence of options, used by the compiler in implementation-defined ways.

```
base_domain_declaration ::= [ extended ] base domain Ada_identifier_1
                                [ ( base_domain_parameter_list ) ]
                            is
                                patterns
                                options
                            end [ Ada_identifier_2 ] ;

base_domain_parameter_list ::= base_domain_parameter { ; base_domain_parameter }
```

1. If present, *Ada*_identifier_2 shall equal *Ada*_identifier_1. *Ada*_identifier_1 is the *name* of the base domain.

2. The keyword **extended** may appear in a base_domain_declaration only if it also appears in the enclosing module declaration.

4.1.1.1 Base Domain Parameters

```
base_domain_parameter ::=  Ada_identifier : data_class [ := static_expression ]  |
                           map := pos                                            |
                           map := image

data_class ::=  integer     |
                character   |
                fixed       |
                float       |
                enumeration
```

1. The Ada identifiers within the list of base_domain_parameters of a base_domain_declaration are the names of the parameters that may appear in a parameter_association_list within a domain_declaration based on this base domain (see section 4.1.3). The static_expression within a base_domain_parameter, when present, specifies a default value for the parameter. This default value shall be of the correct data_class; that is, in the parameter declaration

   ```
   Id : dcl := expr ;
   ```

   where *dcl* is a data_class, DATACLASS(expr) shall be *dcl*. Further, DATACLASS(Id) is *dcl*, whether or not the initializing expression *expr* is present, and DOMAIN(Id) is NO_DOMAIN.

2. A base domain is classified by its data_class. That is, an enumeration base domain is a base domain whose data_class is **enumeration**, a fixed base domain is a base domain whose data_class is **fixed**, etc.

3. Every enumeration base domain has two predefined parameters: **enumeration** and **map**. These parameters are special in that the values that are assigned to them by a domain

declaration (see section 4.1.3) are not of any of the data classes listed above. The value of an **enumeration** parameter is an enumeration_reference (see section 3.4); the value of **map** is a database_mapping (see section 4.1.3). A base domain declaration may explicitly declare a **map** parameter for the purpose of assigning a default mapping. An enumeration base domain shall not redefine the predefined base_domain_parameter **enumeration**.

There are two possible default mappings: **pos** and **image**. The value **pos** specifies that the Ada predefined attribute function 'POS of the Ada type corresponding to the enumeration_reference, which is the **enumeration** parameter value in the domain declaration, shall be used to translate enumeration literals to their database encodings. Similarly for **image** and the 'IMAGE attribute. See annex A of **[Ada]** and sections 4.1.3 and 4.3 of this document.

4. Every fixed base domain has a predefined parameter **scale** whose value is an integer of an implementation defined range (see **[SQL]**, section 5.5). A fixed base domain shall not redefine the predefined base_domain_parameter **scale**.

5. Every character base domain has a predefined parameter **length** whose value is an integer of an implementation defined range (see **[SQL]**, section 5.5). A character base domain shall not redefine the predefined base_domain_parameter **length**.

4.1.1.2 Base Domain Patterns

The patterns portion of a base domain declaration forms a template for the generation of Ada text, which forms the Ada semantics of domains based on the given base domain.

```
patterns ::= { pattern }

pattern ::=     domain_pattern          |
                subdomain_pattern       |
                derived_domain_pattern

domain_pattern ::=      domain pattern is pattern_list
                        end pattern ;

subdomain_pattern ::=   subdomain pattern is pattern_list
                        end pattern ;

derived_domain_pattern ::=  derived domain pattern is pattern_list
                            end pattern ;

pattern_list ::= pattern_element { pattern_element }

pattern_element ::= character_literal
```

Patterns are used to create the Ada constructs that implement the Ada semantics of a domain, subdomain, or derived domain declaration (see sect 4.1.3). Patterns are considered templates; parameters within a pattern are replaced by the values assigned to them either in the domain declaration, by inheritance, or by default.

For a parameter to be recognized as such in a pattern, it is enclosed in square brackets ([,]). For the purpose of pattern substitution, a base domain may use a parameter **self**. When a pattern is instantiated, **self** is the name of the domain or subdomain being declared. A base domain may use a parameter **parent** for the purpose of pattern substitution in a subdomain_pattern or a

derived_domain_pattern. When such a pattern is instantiated, **parent** is the name of the parent domain (see section 4.1.3).

Within a given character_literal of a pattern, a substring contained in matching curly brackets ({, }) is an optional phrase. Optional phrases may be nested. An optional phrase appears in the instantiated template if all parameters within the phrase have values assigned by a domain declaration; the phrase does not appear when none of the parameters within the phrase has an assigned value. If some but not all parameters within an optional phrase have values assigned by a given domain declaration, the declaration is in error.

4.1.1.3 Base Domain Options

```
options ::= { options }

option ::=      fundamental    |
                for word_list use pattern_list ;   |
                for word_list use predefined ;

fundamental ::= for not null type name use pattern_list ;          |
                for null type name use pattern_list ;              |
                for data class use data_class ;                    |
                for dbms type use dbms_type [ pattern_list ] ;     |
                for conversion from type to type use converter ;

dbms_type ::=   int                |
                integer            |
                smallint           |
                real               |
                double precision   |
                char               |
                character          |
                implementation defined

type ::= dbms | not null | null

converter ::=   function pattern_list    |
                procedure pattern_list   |
                type mark

word_list ::=   context clause           |
                null value               |
                null_bearing assign      |
                not_null_bearing assign
```

Options are used to define aspects of base domains that are essential to the declaration of domains within the SAMeDL. The fundamental options are required. Implementations may add options beyond those given above. The meanings of the fundamental options are given by the following list.

1. The **null** and **not null** type names are the targets of the function AdaTYPE. They are the names of the types of parameters and parameter components in Ada procedures. See sections 5.6 and 5.7.

2. The **data class** option specifies the data class (see section 4.1.1.1) of all objects of any domain based on this base domain. If *BD* is a base domain to which the data class *dc* is

assigned by an option in its definition, and if *D* is a domain based directly or indirectly (see section 4.1.3) on *BD*, then DATACLASS(D) = dc. The data class governs the use of literals with such objects (see sections 5.8 and 5.10).

3. The **dbms type** of a base domain is the *SQL*_data_type (see **[SQL]** section 5.5) to be used when declaring parameters of the concrete interface (SQL module) for all objects of domains based directly or indirectly on the base domain. See sections 5.6, 5.7, and 5.8 of this document. If the dbms type of a base domain is implementation defined, the keyword **extended** shall appear in the declaration of the base domain.

4. An operand of the **conversion** option is a means of converting non-null data between objects of the not null-bearing type, the null-bearing type (see section 4.1.3) and dbms type associated with a domain. A method shall be a function, a procedure, an attribute of a type, or a type conversion. A means of determining the identity of these methods shall appear in the options of a base domain. The identity of a method may be given as a pattern containing parameters.

   However, enumeration domains do not have converters between the dbms type and the not null-bearing type, as the **map** parameter predefined for all enumeration domains describes a conversion method between enumeration and database representations of non-null data. The method is the application, as appropriate of the function described by the database_mapping that is the operand of the map parameter association (see sections 4.1.1.1 and 4.1.3).

Additional, implementation-defined options are used to provide information to the SAMeDL Compiler that is not provided via the fundamental options. The Intermetrics SAMeDL Compiler makes use of 4 additional options, described by the *word_list* grammar. These options are all required. The meanings of these options are given by the following list.

1. The **context clause** option specifies the WITH and/or USE clauses required by packages that declaredomains using the base domain. Each base domain must rely on at least one of the SAMeDL standard packages (SQL_Char, SQL_Int, etc.), so this option is required. The context clauses are specified as patterns, and the patterns must not include references to [self], [parent], or any of the base domain parameters. This option must appear once and only once for each base domain declaration.

2. The **null value** option provides the SAMeDL compiler with a pattern that can be used as a null value for all domain declarations based on the base domain declaration.

3. The **null_bearing assign** option designates a function for assigning an object of the base domain's null-bearing type to another object of the null-bearing type. Either a pattern or the word **predefined** may be used to express the conversion function. Use of the word **predefined** indicates that the standard operator := should be used to perform the conversion.

4. The **not_null_bearing assign** option designates a function for assigning an object of the base domain's not-null type to another object of the not-null type. Either a pattern or the word **predefined** may be used to express the conversion function. Use of the word **predefined** indicates that the standard operator := should be used to perform the conversion.

4.1.2 The SAME Standard Base Domains

The predefined definitional module, SAMeDL_Standard, contains the declarations of the predefined SAME Standard Base Domains: SQL_Int, SQL_Smallint, SQL_Char, SQL_Real, SQL_Double_Precision, SQL_Enumeration_as_Char, and SQL_Enumeration_as_Int. The text of SAMeDL_Standard appears in Appendix A.

4.1.3 Domain and Subdomain Declarations

domain_declaration ::= **domain** *Ada*_identifier **is new** bas_dom_ref [**not null**]
[(parameter_association_list)] ;

subdomain_declaration ::= **subdomain** *Ada*_identifier **is** dom_ref [**not null**]
[(parameter_association_list)] ;

dom_ref ::= domain_reference | subdomain_reference

bas_dom_ref ::= dom_ref | base_domain_reference

parameter_association_list ::= parameter_association { , parameter_association }

parameter_association ::= *Ada*_identifier => static_expression |
map => database_mapping |
enumeration => enumeration_reference |
scale => static_expression |
length => static_expression

database_mapping ::= enumeration_association_list | **pos** | **image**

enumeration_association_list ::= (enumeration_association { , enumeration_association })

enumeration association ::= enumeration_literal => database_literal

1. Consider the domain declaration:

   domain DD **is new** EE ....

   a. If EE is a base_domain_reference, then EE is said to be the base domain of DD.

   b. Otherwise, EE is a domain_reference or subdomain_reference, the base domain of DD is defined to be the base domain of EE, DD is said to be derived from EE, and EE is said to be the parent of DD.

2. Similarly, in the subdomain declaration

   subdomain FF **is** GG

   the base domain of FF is defined as the base domain of GG, FF is said to be a subdomain of GG, and GG is said to be the parent of FF.

3. The database type of a domain *D*, denoted as DBMS_TYPE(*D*), is the value, appropriately parameterized, of the **for dbms type** option from the base domain of D. See section 4.1.1.3.

4. The data class of a domain *D*, denoted DATACLASS(*D*), is the data class of its base domain, the value of the **for data class** option. A domain is numeric if its data class is numeric.

5. Except for **scale, enumeration, length,** and **map,** an *Ada*_identifier within a parameter_association shall be the name of a base_domain_parameter in the declaration of the base domain of the domain or subdomain being declared. See section 4.1.1.1.

6. A domain or subdomain *D* is said to *assign the expression E* to the parameter *P*, if

   a. the parameter_association *P* => *E* appears in the declaration of *D*; or

   b. (a) does not hold, *D* is a subdomain or a derived domain, and the parent domain assigns the expression *E* to the parameter *P;* or

   c. (a) and (b) do not hold and in the base_domain_declaration for the base domain of *D*, the base_domain_parameter

      ```
      P : class := E
      ```

      appears.

   In all cases, DATACLASS(*E*) shall be DATACLASS(*P*) as defined by the declaration of the base domain. See section 4.1.1.1.

7. If a domain *D* assigns the expression *E* to a parameter *P*, then

   - DOMAIN(D.P) = NO_DOMAIN
   - DATACLASS(D.P) = DATACLASS(E)
   - LENGTH(D.P) = LENGTH(E)
   - SCALE(D.P) = SCALE(E)

8. A domain_declaration shall assign an expression to each base_domain_parameter that appears in any non-optional phrase

   - of the base domain's domain_pattern, if the declaration is not declaring a derived domain;
   - of the base domain's derived_domain_pattern, if the declaration is the declaration of a derived domain.

   Similar rules govern subdomain_declarations and subdomain_patterns. See section 4.1.1.2.

9. The scale of a domain *D*, denoted SCALE(*D*), is defined by

   - if *D* is not a numeric domain, SCALE(*D*) = NO_SCALE;
   - if *D* is an integer domain, SCALE(*D*) = 0;
   - if *D* is a float domain, SCALE(*D*) = a value greater than the scale of any non-float domain or object;

- if *D* is a fixed domain, SCALE(*D*) = the value assigned by *D* to the **scale** base_domain_parameter.

The value assigned to the **scale** parameter in the declaration of a fixed domain shall be an integer from an implementation defined range.

10. The length of a domain *D*, denoted LENGTH(*D*), is defined by

    - if *D* is not a character domain, LENGTH(*D*) = NO_LENGTH;
    - if *D* is a character domain, LENGTH(*D*) = the value assigned by *D* to the **length** base_domain_parameter.

    The value assigned to the **length** parameter in the declaration of a character domain shall be an integer from an implementation defined range.

11. Any domain_declaration or subdomain_declaration of an enumeration domain shall assign an enumeration_reference to the base_domain_parameter **enumeration** and a database_mapping to the base_domain_parameter **map**. If the **map** parameter is assigned an enumeration_association_list, then

    - Each enumeration_literal within the enumeration referenced by the enumeration_reference given by the **enumeration** parameter shall appear as the enumeration_literal of exactly one enumeration_association.
    - No database_literal shall appear in more than one enumeration_association.

    Note: These constraints ensure that the database_mapping is an invertible (i.e., one-to-one) function. That function is used for both compile time and runtime data conversions. See sections 4.1.1.3 and 4.3.

12. Let *D* be an enumeration domain or subdomain declaration and let *En* be the name of the enumeration referenced by the value assigned by *D* to the **enumeration** base_domain_parameter. *D* is said to *assign the expression E* to the enumeration literal *El*, if *D* assigns the database_mapping *M* as the value of the **map** base_domain_parameter and *M*

    - is **pos**, and *E* = *En'Pos(El)*, or
    - is **image**, and *E* = *En'Image(El)*, or
    - is an enumeration_association_list containing an enumeration_association of the form *El* => *E*

    See section 4.1.6.

13. The database_mapping of an enumeration domain or subdomain declaration *D* should preserve the ordering implied by that domain's enumeration_reference *ER*. That is, if L_1 and L_2 are enumeration literals of *ER* such that L_1 occurs before L_2 in *ER's* enumeration_literal_list, then the value assigned to L_1 by *D* should be less than the value assigned to L_2 by *D*.

14. A domain or subdomain is said to be *not null only* if it or any of its parent domains is declared with the **not null** phrase. In that case no object of the domain can contain the null value.

Ada Semantics

An instantiation of a pattern defined for the base domain of the domain being declared, as described in section 4.1.1.2, shall appear within the Ada package specification corresponding to the module within which the domain_declaration appears. If in the domain declaration:

```
domain DD is new EE ...
```

EE is a base_domain_reference, then the domain_pattern is instantiated; if EE is a domain_reference, the derived_domain_pattern is instantiated; for the subdomain declaration

```
subdomain FF is GG ...
```

the subdomain_pattern is used.

Examples:

The following examples illustrate the declaration of domains and have been annotated with references to the appropriate clauses of the language definition. The base domains used in these examples exist in the predefined definitional module SAMeDL_Standard, which appears in Appendix A. The constant Max_SQL_Int is declared in the predefined definitional module SAMeDL_System (see Appendix B). Both SAMeDL_Standard and SAMeDL_System are assumed to be visible, as is the enumeration declaration *Colors* (see section 4.1.6).

```
domain Weight_Domain is new SQL_Int ( -- 4.1.3: #1a
     First    => 0,                   -- 4.1.3: #5 and #8
     Last     => Max_SQL_Int);        -- 4.1.3: #5 and #8

domain City_Domain is new SQL_Char ( -- 4.1.3: #1a
     Length => 15);                  -- 4.1.3: #5 and #10

domain Color_Domain is new SQL_Enumeration_As_Char ( -- 4.1.3: #1a
     enumeration  => Colors,                         -- 4.1.3: #11
     map          => Image);                         -- 4.1.3: #11 and #12

domain Auto_Weight is new Weight_Domain (    -- 4.1.3: #1b
     Last    => 10000);                      -- 4.1.3: #5

subdomain Auto_Part_Weight is Auto_Weight (  -- 4.1.3: #2
     Last    => 2000);                       -- 4.1.3: #5 and #8
```

The declarations produce the following Ada code:

```
-- the Ada code below is the instantiation of the domain pattern
--      from the base domain SQL_Int

type Weight_Domain_Not_Null is new SQL_Int_Not_Null
     range 0 .. implementation_defined;
type Weight_Domain_Type is new SQL_Int;
```

```
package Weight_Domain_Ops is new SQL_Int_Ops (
      Weight_Domain_Type, Weight_Domain_Not_Null);

-- the Ada code below is the instantiation of the domain pattern
--      from the base domain SQL_Char

type City_DomainNN_Base is new SQL_Char_Not_Null;
subtype City_Domain_Not_Null is City_DomainNN_Base (1 . . 15);
type City_Domain_Base is new SQL_Char;
subtype City_Domain_Type is City_Domain_Base (City_Domain_Not_Null'Length);
package City_Domain_Ops is new SQL_Char_Ops
      (City_Domain_Base, City_DomainNN_Base);


-- the Ada code below is the instantiation of the domain pattern
--      from the base domain SQL_Enumeration_As_Char

type Color_Domain_not_null is new Colors;
package Color_Domain_Pkg is new SQL_Enumeration_Pkg (Color_Domain_not_null);
type Color_Domain_Type is new Color_Domain_Pkg.SQL_Enumeration;


-- the Ada code below is the instantiation of the derived domain pattern
--      from the base domain SQL_Int

type Auto_Weight_Not_Null is new Weight_Domain_Not_Null
      range Weight_Domain_Not_Null'First . . 10000;
type Auto_Weight_Type is new Weight_Domain_Type;
package Auto_Weight_Ops is new SQL_Int_Ops (
      Auto_Weight_Type, Auto_Weight_Not_Null);


-- the Ada code below is the instantiation of the subdomain pattern
--      from the base domain SQL_Int

subtype Auto_Part_Weight_Not_Null is Auto_Weight_Not_Null
      range Auto_Weight_Not_Null'First . . 2000;
type Auto_Part_Weight_Type is new Auto_Weight_Type;
package Auto_Part_Weight_Ops is new SQL_Int_Ops (
      Auto_Part_Weight_Type, Auto_Part_Weight_Not_Null);
```

4.1.4 Constant Declarations

```
constant_declaration ::= constant Ada_identifier [ : domain_reference ]
                              is static_expression ;

static_expression ::= value_expression
```

A static expression is a value expression (see section 5.10) whose value can be calculated at compile time; i.e., whose leaves are all either literals or constants.

Let *K* denote the constant declaration

```
constant C [: D ] is E ;
```

1. DATACLASS(K) is DATACLASS(E), the data class of the static expression E.

2. If DATACLASS(K) is **enumeration**, then D shall be present in the constant declaration and shall name an enumeration domain of which the static expression E is an enumeration literal.

3. If DATACLASS(K) is **character**, then D shall be present.

4. If the domain_reference D is not present, then

   a. C is a *universal* constant of type DATACLASS (K).

   b. AdaTYPE(K) is an anonymous type, *universal_T*, where T is DATACLASS(K).

   c. if DATACLASS(K) is numeric, then SCALE(K) = SCALE(E).

   d. DOMAIN(K) = NO_DOMAIN.

5. If the domain_reference D is present, then

   a. DOMAIN(K) = D and E shall conform to D.

   b. If DATACLASS(K) is numeric, then SCALE(K) = SCALE(D), and SCALE(E) shall not exceed SCALE(D).

   c. If DATACLASS(K) is character, then LENGTH(K) = LENGTH(D) and LENGTH(E) shall not exceed LENGTH(D).

   d. AdaTYPE(K) is defined as the type name within D designated as not null bearing.

Ada Semantics

Let VALUE represent the function which calculates the value of a static_expression. Let *SE* be a static_expression. VALUE(SE) is given recursively as follows:

1. If SE contains no operators, then

   a. If SE is a database_literal, then VALUE(SE) = SE.

   b. If SE is an enumeration_literal of domain D, and D assigns expression E to that enumeration literal, then VALUE(SE) = E.

   c. If SE is a reference to the constant whose declaration is given by

      constant C [: D] **is** E ;

      then VALUE(SE) = VALUE(E).

   d. If SE is a reference to a parameter P from domain D, and D assigns the expression E to P, then VALUE(SE) = VALUE(E).

2. If SE is $D(SE_1)$, where D is a domain name, then $VALUE(D(SE_1)) = VALUE(SE_1)$.

3. If SE is $+SE_1$ (or $-SE_1$), then $VALUE(SE) = +VALUE(SE_1)$ (or $-VALUE(SE_1)$)

4. If SE is SE_1 *op* SE_2 where *op* is an arithmetic operator, then VALUE(SE) = VALUE(SE_1) *op* VALUE(SE_2) where *op* is evaluated according to the rules of SQL.

5. If SE is (SE_1) then VALUE(SE) = (VALUE(SE_1)).

Again, let *K* denote the constant declaration

constant C [: D] **is** E ;

Let *Q* be the Ada representation of VALUE(E). Then the Ada library package specification corresponding to the module in which the constant declaration K above appears shall have an Ada constant declaration of the form

C : constant [AdaTYPE (K)] := Q ;

The type designator AdaTYPE(K) is omitted from this declaration if it is an anonymous type.

Examples:

The following SAMeDL constant declarations

```
constant Zero is 0;                  -- a named number of type universal_int
constant Val : ValDomain is 1;       -- a constant value of ValDomain type
```

will produce the following Ada code:

```
Zero : constant := 0;
Val  : constant ValDomain_not_null := 1;
```

4.1.5 Record Declarations

record_declaration ::= **record** *Ada*_identifier_1 [named_phrase] **is**
component_declarations
end [*Ada*_identifier_2] ;

named_phrase ::= **named** *Ada*_identifier

component_declarations ::= component_declaration { component_declaration }

component_declaration ::= component { , component } : domain_reference [**not null**] ;

component ::= component_name [**dblength** [named_phrase]]

component_name ::= *Ada*_identifier

If present, *Ada*_identifier_2 shall be equal to *Ado*_identifier_1. *Ada*_identifier_1 is the *name* of the record.

Let *R* be a record declaration. Define AdaNAME(R) to be

1. The alias *N*, if the named_phrase **named** *N* appears in the declaration.

2. *Row*, otherwise.

Note: AdaNAME(R) is the default for the name of the row record formal parameter in the parameter profile of any procedure which uses the declaration *R*. See Sections 5.2, 5.5 and 5.9.

Ada Semantics

The Ada library unit package specification corresponding to the module within which the record_declaration *R* appears shall have an Ada record type declaration (called R_{Ada}) defined as follows:

1. The name of the record type R_{Ada} shall be *Ada*_identifier_1.

2. For some integer *k*, let the component_declarations of *R* be given by the sequence

   components$_i$: D_i [**not null**$_i$]

   for $1 \leq i \leq k$, where components$_i$ is given by the sequence

   C_{ij} [**dblength**$_{ij}$ [**named** N_{ij}]]

   where $1 \leq j \leq m_i$ for some integer m_i. R_{Ada} shall be equivalent, in the sense of **[Ada]** sections 3.2.10 and 3.7.2, to a record type whose components are given by the sequence

   $COMP_{ij}$ [$DBleng_{ij}$]

   where *i* and *j* are bound as before and $COMP_{ij}$ is given by

   C_{ij} : T_i ;

   where T_i is an Ada type name determined to be:

   a. The not null-bearing type name within the domain D_i, if either D_i is a not null only domain or **not null**$_i$ is present in *R*;

   b. Otherwise the null-bearing type name within the domain D_i.

   If the optional **dblength**$_{ij}$ phrase is specified, then $DBleng_{ij}$ appears and takes the form

   $DBLngNAME_{ij}$: Ada_Indicator_Type ;

   where $DBLngNAME_{ij}$ is N_{ij} if N_{ij} appears and is C_{ij}_DbLength, otherwise; Ada_Indicator_Type is the type SQL_Standard.Indicator_Type (see **[ESQL] 8.a.3**).

Examples:

The following SAMeDL record declaration

```
record Parts_Row_Record_Type named Parts_Row_Record is
    Part_Number      : Pno_Domain not null;
```

```
        Part_Name          : Pname_Domain;
        Color              : Color_Domain;
        Weight_In_Ounce    : Weight_Domain;
        City               : City_Domain;
    end Parts_Row_Record_Type;
```

will produce the following Ada code:

```
    type Parts_Row_Record_Type is record        -- Ada Semantics #1
        Part_Number    : Pno_Domain_not_null;   -- Ada Semantics #2
        Part_Name      : Pname_Domain_Type;
        Color          : Color_Domain_Type;
        Weight         : Weight_Domain_Type;
        City           : City_Domain_Type;
    end record;
```

4.1.6 Enumeration Declarations

Enumerations are used to declare sets of enumeration literals for use in enumeration domains and status maps (see sections 4.1.3 and 4.1.8).

enumeration_declaration ::= **enumeration** *Ada*_identifier_1 **is** (enumeration_literal_list) ;

enumeration_literal_list::= enumeration_literal { , enumeration_literal }

1. *Ada*_identifier_1 is the name of the enumeration.

2. Each identifier within an enumeration_literal_list is said to be an enumeration literal of the enumeration. The enumeration_declaration is considered to declare each of its enumeration_literals. An enumeration_literal may appear in multiple enumeration declarations.

Ada Semantics

There shall be, within the Ada package specification corresponding to the module within which an enumeration appears, an Ada enumeration type declaration of the form

type *Ada*_identifier_1 **is** (enumeration_literal_list) ;

Note: Ada character literals shall not be used in enumerations.

Examples:

The following are examples of SAMeDL enumeration declarations.

```
enumeration Sizes is (small, medium, large, x_large);
enumeration Sex is (M, F);
enumeration Operation_Status is (Disk_Error,
    Data_Conversion_Error, Invalid_SQL_Statement, Not_Found);
enumeration Colors is (Purple, Blue, Green, Yellow, Orange, Red, Black, White) ;
```

For the declarations above, the following Ada code would be produced:

```
type Sizes is (small, medium, large, x_large);
type Sex is (M, F);
type Operation_Status is (Disk_Error,
     Data_Conversion_Error, Invalid_SQL_Statement, Not_Found);
type Colors is (Purple, Blue, Green, Yellow, Orange, Red, Black, White) ;
```

4.1.7 Exception Declarations

exception_declaration ::= **exception** *Ada*_identifier ;

*Ada*_identifier is the *name* of the exception.

Ada Semantics

There shall be, within the Ada package specification corresponding to the module within which an exception declaration appears, an exception declaration of the form

*Ada*_identifier_1 : **exception**;

Examples:

The following are examples of exception declarations.

```
exception Data_Definition_Does_Not_Exist;
exception Insufficient_Privilege;
```

The above declarations produce the following Ada code:

```
Data_Definition_Does_Not_Exist : exception;
Insufficient_Privilege : exception;
```

4.1.8 Status Map Declarations

The execution of any procedure (see sections 3.7, 5.2, and 5.5) causes the execution of an SQL procedure. That execution causes a special parameter, called the SQLCODE parameter, to be set to a status code that either indicates that a call of the procedure completed successfully or that an exception condition occurred during execution of the procedure. Status maps are used within abstract modules to process the status data in a uniform way. Each map defines a partial function from the set of all possible SQLCODE values onto (1) enumeration literals of an enumeration and (2) raise statements. ***Note:*** The function is DBMS specific in that SQLCODE values are not specified by standard SQL, whereas the enumeration type and exceptions are not specific to any DBMS.

```
status_map_declaration ::=   status Ada_identifier_1
                                 [ named_phrase ]
                                 [ uses target_enumeration ]
                             is ( sqlcode_assignment { , sqlcode_assignment ) ;

target_enumeration ::= enumeration_reference | boolean
```

```
sqlcode_assignment ::=  static_expression_list => enumeration_literal    |
                        static_expression_list => raise exception_reference

static_expression_list ::=  static_expression { , static_expression } |
                            static_expression .. static_expression
```

1. *Ada*_identifier_1 is the *name* of the status map.

2. A target enumeration of **boolean** is a reference to the predefined Ada enumeration type Standard.Boolean.

3. If the optional **uses** clause is not present, then only sqlcode_assignments that contain **raise** shall be present in the status_map_declaration.

4. Every *Ada*_enumeration_literal within an sqlcode_assignment shall be an *Ada*_enumeration_literal within the enumeration referenced by the target_enumeration.

5. If E => L (or E => **raise** X) is an sqlcode_assignment then

   - DATACLASS(E) = **Integer** .
   - If E' => L' (or E' => **raise** X') is any other sqlcode_assignment within the status_map_declaration, then E and E' shall not evaluate to the same integer.

Note: An sqlcode_assignment takes the form of a list of alternatives as found in Ada case statements, aggregates, and representation clauses. The **others** choice is not valid for sqlcode_assignments, however.

Note: SAMeDL_Standard contains the definition of a status map Standard_Map, defined as follows:

```
status Standard_Map named Is_Found uses boolean is
    (0 => True, 100 => False);
```

Standard_Map is the status map for those fetch statements that appear in cursor declarations by default (see section 5.5). It signals end of table by returning false.

Examples:

```
status Operation_Map named Result_Of_Operation
    uses Operation_Status is (
        -600 .. -699        => Disk_Error,
        -500 .. -599        => Data_Conversion_Error,
        -300 .. -499        => Invalid_SQL_Statement,
        -101, -110, -113    => raise Data_Definition_Does_Not_Exist,
        -25                 => raise Insufficient_Privilege,
         100                => Not_Found);
```

4.2 Schema Modules

Schema modules contain the database description.

```
schema_module ::=   [ context ]
                    [ extended ]
                    schema module SQL_identifier_1 is
                        { schema_element }
                    end [ SQL_identifier_2 ] ;

schema_element ::=  table_definition            |
                    view_definition             |
                    SQL_privilege_definition    |
                    extended_schema_element

SQL_privilege_definition ::= (see [SQL] 6.10)
```

1. If present, *SQL*_identifier_2 shall be equal to *SQL*_identifier_1. *SQL*_identifier_1 is the *name* of the schema_module.

2. *SQL*_identifier_1 shall be different from any other schema module name.

3. An extended_schema_element may appear in a schema_module only if the keyword **extended** appears in the associated schema module declaration.

4.2.1 Table Definitions

Table definitions are analogous to SQL table declarations in that they provide information concerning the underlying structure of a database table within a schema.

```
table_definition ::=    [ extended ] table SQL_identifier_1 is
                            table_element { , table_element }
                        end [ SQL_identifier_2 ] ;

table_element ::=       column_definition           |
                        table_constraint_definition |
                        extended_table_element

column_definition ::=   SQL_column_name [ SQL_data_type ]
                            [ SQL_default_clause ]
                            [ column_constraint ] : domain_reference

SQL_default_clause ::= (see [SQL] 6.4)

column_constraint ::=       not null| SQL_unique_specification      |
                            SQL_reference_specification             |
                            check ( search_condition )

SQL_unique_specification ::= (see [SQL] 6.6)

SQL_reference_specification ::= (see [SQL] 6.7)

table_constraint_definition ::= SQL_unique_constraint_definition        |
                                SQL_referential_constraint_definition   |
                                check_constraint_definition

SQL_unique_constraint_definition ::= (see [SQL] 6.6)

SQL_referential_constraint_definition ::= (see [SQL] 6.7)
```

check_constraint_definition ::= **check** (search_condition)

1. If present, *SQL*_identifier_2 shall be equal to *SQL*_identifier_1. *SQL*_identifier_1 is the *name* of the table and the table_definition.

2. The name of the table_definition must be different from the name of any other table_definition or view_definition within the enclosing schema_module.

3. A table_definition shall contain at least one column_definition.

4. Every *SQL*_column_name shall be distinct from every other *SQL*_column_name within the enclosing table_definition.

5. If the column_constraint is absent from a column_definition, then the domain_reference shall not be to a not null only domain.

6. For the semantics of **not null**, see **[SQL]**, sections 6.3 and 6.6; for the semantics of **check**, see **[SQL]**, sections 6.3 and 6.8.

7. Suppose that column_definition *CD* is of the form

   CN [DT] [DC] [CC] : D;

   a. The domain of *CD*, denoted DOMAIN(CD), is *D*. If *DT* is present, then conversion between DBMS_TYPE(D) (see section 4.1.3) and *DT* shall be legal in both directions by the rules of SQL (**[SQL]**, section 8.6, syntax rule 3; section 8.7, syntax rule 6; etc.) unless DBMS_TYPE(D) is an implementation defined dbms_type (see section 4.1.1.3), in which case both conversions must be legal by the implementation defined rules.

   b. Define DATACLASS(CD) as DATACLASS(D).

   c. Define LENGTH(CD) as LENGTH(D)

   d. Define SCALE(CD) as SCALE(D).

8. If **extended** appears in a table_definition then **extended** shall also appear in the associated schema_module declaration.

9. If an extended_table_element appears in a table_definition, then the keyword **extended** shall appear in that table_definition.

4.2.2 View Definitions

view_definition ::= **[extended] view** *SQL*_identifier_1 **as** query_spec
[with check option]
end [*SQL*_identifier_2] ;

query_spec ::= query_specification | extended_query_specification

1. If present, *SQL*_identifier_2 shall be equal to *SQL*_identifier_1. *SQL*_identifier_1 is the *name* of the view and the view_definition.

2. The name of the view_definition shall be different from the name of any other table_definition or view_definition in the enclosing schema module.

3. A query_spec may be an extended_query_specification only if the keyword **extended** appears in the associated view_definition.

Examples:

```
with Def_Mod; use Def_Mod;
schema module Parts_Suppliers_Database is
    -- the Parts table
    table P is                                  -- 4.2.1: #1 and #2

        Pno not null  : Part_Domain,            -- 4.2.1: #3 and #4
        Pname         : Pname_Domain,           -- 4.2.1: #5
        Color         : Color_Domain,
        Weight        : Weight_Domain,
        City          : City_Domain,
        unique (Pno)

    end P;

    -- the Suppliers table
    table S is                                  -- 4.2.1: #1 and #2

        Sno not null  : Sno_Domain,             -- 4.2.1: #3 and #4
        Sname         : Sname_Domain,           -- 4.2.1: #5
        Sstatus       : Sstatus_Domain,         -- "Status" is a reserved word
        City          : City_Domain,
        unique (Sno)

    end S;


    -- the Orders table
    table SP is                                                 -- 4.2.1: #1 and #2

        Sno character(5) not null    : Sno_Domain,              -- 4.2.1: #3 and #4
        Pno character(6) not null    : Pno_Domain,
        Qty integer                  : Quantity_Domain,         -- 4.2.1: #5
        unique (Sno, Pno)

    end SP;

    -- the Part_Number_City view
    view Pno_City as                        -- 4.2.2: #1 and #2

        select distinct Pno, City
        from SP, S
            where SP.Sno = S.Sno

    end Pno_City;

end Parts_Suppliers_Database;
```

4.3 Data Conversions

The procedures that are described in an abstract module (see Chapter 5) transmit data between an Ada application and a DBMS. Those data undergo a conversion during the execution of those procedures. Constants and enumeration literals used in statements are replaced by their database representation in the form of the statement in the concrete module. This process occurs at module compile time. Both processes are described in this section.

Execution Time Conversions

The execution time conversions check for and appropriately translate null values; for not null values, the conversion method identified by the appropriate base domain declaration (see section 4.1.1.3).

Input parameter conversion rule. If the type of an input parameter is null-bearing, then in the corresponding SQL procedure there is an associated *SQL*_parameter_specification to which an *SQL*_indicator_parameter has been assigned (see sections 5.6 and 5.8). If, for any execution of the procedure, the value of the input parameter is null, then the indicator parameter is assigned a negative value (see **[SQL]**, subsection 4.10.2 and section 5.6, general rule 1). Otherwise, the indicator parameter shall be non-negative and the SQL parameter shall be set from the input parameter by the conversion process identified for the base domain. If the type of an input parameter is not null-bearing, the SQL parameter shall be set from the input parameter by the conversion process identified for the base domain (see section 4.1.1.3).

Output parameter conversion rule. For output parameters of procedures containing either **fetch** or **select** statements, this process is run in reverse. Let SP be a select parameter. Then the corresponding SQL procedure has a data parameter and an indicator parameter parameter corresponding to SP (see sections 5.2, 5.5, and 5.7). For any execution of the procedure:

- If the indicator parameter is negative, then
  - -- If the type of the Ada record component $COMP_{Ada}(SP)$ (see section 5.2 and 5.5) is null-bearing, then $COMP_{Ada}(SP)$ is set to the null value; *else*
  - -- If the type of $COMP_{Ada}(SP)$ is not null-bearing, the exception SAMeDL_Standard.Null_Value_Error is raised.
- If the indicator parameter is non-negative, then the value of $COMP_{Ada}(SP)$ is set from the value of the SQL data parameter by the conversion process identified for the base domain (see section 4.1.1.3). If the record component $DBLeng_{Ada}(SP)$ is present (see section 5.2 and 5.4), then it is set to the value of the indicator parameter.

Compile Time Conversions

The SQL semantics of constants, domain parameters, and enumeration literals (and constants that evaluate to enumeration literals) used in value lists of insert statements (see section 5.8) and value expressions (see section 5.10) require that they be replaced in the generated SQL code by representations known to the DBMS. For enumeration literals, the enumeration mapping is used (see sections 4.1.1.1, 4.1.1.3, and 4.1.3).

Let *V* be an identifier. If *V* is not a reference to a constant or an enumeration literal, then *V* is not static and undergoes no compile time conversion.

If *V* is a reference to

- a constant declared by

  constant C [: D] **is** E ;

- a domain parameter *param* of domain D, and D assigns the expression E to *param* (see section 4.1.3)

- or an enumeration literal *El* from enumeration domain D (see sections 5.3, 5.8, 5.10, and 5.11), and D assigns the expression E to *V*,

then *V* is replaced by the static expression $SQL_{VE}(E)$ (see section 5.10).

Chapter 5 Abstract Module Description Language

5.1 Abstract Modules

```
abstract_module ::= [ context ]
                    [ extended ]
                    abstract module Ada_identifier_1 is
                        authorization schema_reference
                        { definition }
                        { procedure_or_cursor }
                    end [ Ada_identifier_2 ] ;

procedure_or_cursor ::= cursor_declaration | procedure_declaration
```

1. If present, *Ada*_identifier_2 shall be equal to *Ada*_identifier_1. *Ada*_identifier_1 is the *name* of the abstract module.

2. No two of the items (that is, procedures, cursors, and definitions) declared within an abstract module shall have the same name.

3. For the meaning of "**authorization** schema_reference", see **[SQL]**.

4. A procedure_or_cursor may be an extended procedure or an extended cursor only if the keyword **extended** appears in the abstract module declaration.

Ada Semantics

For each abstract module within a compilation unit there is a corresponding Ada library unit package the name of which is the name of the abstract module, that is *Ada*_identifier_1. The Ada construct giving the Ada semantics of each procedure, cursor, or definition within an abstract module is included within the specification of that library unit package.

SQL Semantics

There is an SQL module associated with each abstract module that gives the SQL semantics of the abstract module. The name of the SQL module is implementation defined. The language clause of the SQL module shall specify Ada. The module authorization clause is implementation defined.

5.2 Procedures

This section discusses procedures which are not associated with a cursor. Cursor procedures are discussed in Section 5.5.

For every procedure declared within an abstract module there is an Ada procedure declared within the library unit package specification corresponding to that abstract module and an SQL procedure declared within the corresponding SQL module (see Section 5.1). A call to the Ada procedure results in the execution of the SQL procedure.

```
procedure_declaration ::= [ extended ]
                          procedure Ada_identifier_1
                              [ input_parameter_list ]
```

```
                    is
                        statement
                        [ status_clause ]
                    ;

statement ::= commit_statement          |
              delete_statement          |
              insert_statement_values   |
              insert_statement_query    |
              rollback_statement        |
              select_statement          |
              update_statement          |
              extended_statement
```

1. *Ada*_identifier_1 is the *name* of the procedure.

2. An input_parameter_list may appear only in conjunction with statements which take input parameters or with extended statements. In particular, such lists may not appear in procedures containing a commit, rollback, or insert values statement.

3. A statement may be an extended_statement only if the keyword **extended** appears in the procedure declaration.

Ada Semantics

Each procedure declaration P shall be assigned an Ada procedure declaration P_{Ada} in a manner which satisfies the following constraints:

- If P is declared within the declaration of an abstract module M, then P_{Ada} is declared directly within the library unit package specification M.

- The simple name of P_{Ada} is the name of P.

The parameter profile of the Ada procedure is defined as follows:

1. If the statement within the procedure is either a delete, insert_statement_query, select or update statement, then let there be k input parameters (for some $k \geq 0$) in the input parameter list given by INP_1, INP_2, ..., INP_k. Then the i^{th} parameter in the *Ada*_formal_part of P_{Ada} denoted $PARM_{Ada}(INP_i)$ for i<=k, takes the following form (see Section 5.6):

   $AdaNAME(INP_i)$: **In** $AdaTYPE(INP_i)$

2. If the statement within the procedure is a select_statement, then the $(k+1)^{st}$ parameter in the Ada formal part of P_{Ada} is a row record. The mode of the row record parameter shall be **in out**.

   Let *IC* be the into_clause appearing (possibly by assumption, see section 5.3) in the select_statement. Then the name of the row record parameter is $PARM_{Row}(IC)$; the name of the type of that parameter is $TYPE_{Row}(IC)$ (see Section 5.9). If *IC* contains the keyword **new**, then the declarative region containing the declaration of P_{Ada} shall also contain the declaration of $TYPE_{Row}(IC)$.

The names, types and order of the components of the row record parameter are determined from the select_list within the select_statement. Let that list be given by SP_1, SP_2, ..., SP_m. (If the select_list takes the form '*' then assume the transformation described in Section 5.7 has been applied). Then the row record type is equivalent in the sense of **[Ada]**, section 3.2.10 and 3.7.2, to a record whose sequence of components is given by the sequence

$$\ldots \; COMP_{Ada}(SP_i) \; [\; DBleng_{Ada}(SP_i) \;]$$

where $COMP_{Ada}(SP_i)$ is given by

$$AdaNAME(SP_i) \; : \; AdaTYPE(SP_i)$$

provided that $AdaNAME(SP_i)$ and $AdaTYPE(SP_i)$ are defined (see Section 5.7). The record component $COMP_{Ada}(SP_i)$ is otherwise undefined. The record component $DBLeng_{Ada}(SP_i)$ is given by

$$DBLngNAME(SP_i) \; : \; Ada\_Indicator\_Type$$

where Ada_Indicator_Type is the type SQL_Standard.Indicator_Type (see **[ESQL]** section 8.3.a), *provided* that $DBLngNAME(SP_i)$ is defined; otherwise this component is not present.

Note: $COMP_{Ada}(SP_i)$ is undefined only if the i^{th} select parameter is improperly written; whereas $DBLeng_{Ada}(SP_i)$ is undefined only if the i^{th} select parameter does not have a **dblength** phrase (see section 5.7).

3. If the statement within the procedure is an insert_statement_values and it is *not* the case that the insert_value_list is present and consists solely of literals and constants, then the first parameter is a row record. The mode of the record parameter is **in**.

   Let *IC* be the insert_from_clause appearing (possibly by assumption, see section 5.3) in the statement. Then the name of the row record parameter is $PARM_{Row}(IC)$; the name of the type of that parameter is $TYPE_{Row}(IC)$ (see Section 5.9). If *IC* contains the keyword **new**, then the declarative region containing the declaration of P_{Ada} shall also contain the declaration of $TYPE_{Row}(IC)$.

   The names, types and order of the components of the record type are determined from the insert_column_list and insert_value_list. Let C_1, C_2 ..., C_m be the result of insert columns appearing in an insert_column_list such that the corresponding element of the insert_value_list is not a literal or constant reference. Then the row record type is equivalent in the sense of **[Ada]**, section 3.2.10 and 3.7.2, to the record whose i^{th} record component $COMP_{Ada}(C_i)$ for $1 \leq i \leq m$, is given by

   $$AdaNAME(C_i) : AdaTYPE(C_i)$$

   (see Section 5.8).

4. If the statement within the procedure is an extended_statement, see section 3.7; for extended parameter lists, see section 5.6.

5. For all procedures, regardless of statement type, if a status_clause appears in the procedure declaration, then the final parameter is a status parameter of mode **out**. For the name and type of this parameter see Sections 4.1.8 and 5.13.

SQL Semantics

Each procedure declaration P shall be assigned an SQL procedure P_{SQL} within the SQL module for the abstract module in which the procedure appears. P_{SQL} has three parts:

1. An *SQL*_procedure_name . This is implementation defined.

2. A list of *SQL*_parameter_declarations. An SQLCODE parameter is declared for every SQL procedure. Other parameters depend on the type of the statement within the procedure P.

    a. If the statement is a delete, insert_statement_query, select or update statement, then the SQL parameters derived from the input_parameter_list of the procedure, as described in Section 5.6, appear in the parameter declarations of P_{SQL}

    b. If the statement is an insert_statement_values, then the SQL parameters are determined by the subsequence of insert_column_specifications in the insert_column_list whose corresponding entry in the insert_value_list is a column_name (thus not a literal or constant reference). See Section 5.8.

    c. If the statement is a select_statement, then the SQL parameter declarations for P_{SQL} are determined by the select_list of the select_statement, as described in Section 5.7.

    d. If the statement is an extended_statement, see section 3.7.

3. An *SQL*_SQL_Statement (see **[SQL]**, section 7.3). This is derived from the statement in the procedure declaration. See Section 5.3.

Interface Semantics

A call to the Ada procedure P_{Ada} shall have effects which can not be distinguished from the following.

1. The procedure P_{SQL} is executed in an environment in which the values of parameters $PARM_{SQL}(INP)$ and $INDIC_{SQL}(INP)$ (see Section 5.6) are set from the value of $PARM_{Ada}(INP)$ (see Ada Semantics above) according to the rule for input parameters of section 4.3. This holds for every input parameter INP in the input_parameter_list of the procedure or for every column parameter INP in the insert_column_list of an insert_statement_values whose corresponding entry in the insert_column_list is an *SQL*_column_name (thus not a literal or constant_reference). See Section 5.8.

2. Standard post processing, as described in section 3.6, is performed.

3. If the value of the SQLCODE parameter is zero or an implementation defined value which permits the transmission of data (and which is handled by an sqlcode_assignment, see section 3.6), and the statement within the procedure is a select_statement, then the value of the component of the row record parameter $COMP_{Ada}(SP_i)$ and $DBLeng_{Ada}(SP_i)$ are set from the values of the actual parameters

associated with the SQL formal parameters $PARM_{SQL}(SP_i)$ and $INDIC_{SQL}(SP_i)$ (see Section 5.7), according to the rule for output parameters of section 4.3..

Examples:

The following are examples of procedure declarations. The first is a declaration of a procedure with no input parameters.

```
procedure Parts_Suppliers_Commit is
    commit work;
```

The above declaration produces the following Ada procedure declaration in the abstract interface.

```
procedure Parts_Suppliers_Commit;
```

The next procedure declaration contains an input parameter and a status clause.

```
procedure Delete_Parts (
    Input_Pname named Part_Name : Pname_Domain)
is
    delete from P
        where Pname = Input_Pname
        status Operation_Map named Delete_Status
;
```

The above declaration produces the following Ada procedure specification in the abstract interface:

```
procedure Delete_Parts (
    Part_Name     : in Pname_Domain_Type;
    Delete_Status : out Operation_Status);
```

In a somewhat more complex example, involving a row record, the following SAMeDL procedure

```
procedure Insert_Redparts is
    insert into P (
        Pno named Part_Number,
        Pname named Part_Name,
        Color,
        City)
    from Red_Parts
    values (
        Pno,
        Pname,
        'Red',
        City);
```

produces the following Ada declarations:

```
type Insert_Redparts_Row_Type                    -- 5.2, Ada semantics #3, 8.9
     is record
        Part_Number   : Pname_Domain_Type;       -- 4.2.1
        Part_Name     : Pname_Domain_Type;
        City          : City_Domain_Type;
     end record;

procedure Insert_Redparts (Red_Parts : In Insert_Redparts_Row_Type);
```

The color of all parts inserted using the Insert_Redparts procedure will be red. The weight of all such parts will be null. See the examples in section 4.2.2. The number, name and city of those parts are specified at run time.

5.3 Statements

This section describes the concrete syntax of statements other than cursor oriented statements, and defines the text of the SQL statement derived from the text of a SAMeDL statement.

commit_statement ::= **commit work**

rollback_statement ::= **rollback work**

delete_statement ::= **delete from** table_name
[**where** search_condition]

insert_statement_query ::= **insert into** table_name [(*SQL*_insert_column_list)]
query_specification

insert_statement_values ::= **insert into** table_name [(insert_column_list)]
[insert_from_clause] **values** [(insert_value_list)]

update_statement ::= **update** table_name
set set_item { , set_item }
[**where** search_condition]

set_item ::= column_reference = update_value

update_value ::= **null** | value_expression

select_statement ::= **select** [**distinct** | **all**] select_list
[into_clause]
from_clause
[**where** search_condition]
[*SQL*_group_by_clause]
[**having** search_condition]

*SQL*_insert_column_list ::= column_name { , column_name }

*SQL*_group_by_clause ::= **group by** column_reference { , column_reference }

In the following discussion, let *ProcName* be the name of the procedure in which the statement appears.

Ada Semantics

1. If no insert_from_clause appears within an insert_statement_values, then the following clause is assumed:

   from Row : **new** *ProcName*_Row_Type

   If an insert_from_clause which does not contain a record_id appears in an insert_statement_values, the record_id

   : **new** *ProcName*_Row_Type

   is assumed. See Section 5.9.

2. If no into_clause appears within a select_statement, then the following clause is assumed:

   into Row : **new** *ProcName*_Row_Type

   If an into_clause which does not contain a record_id appears in a select_statement, the record_id

   : **new** *ProcName*_Row_Type

   is assumed. See Section 5.9.

3. The following rule applies to both forms of insert statements. If an insert_column_list is not present in such a statement, then a column list consisting of all columns defined for the table denoted by *SQL*_table_name is assumed, in the order in which the columns were declared ([SQL] 8.7.3).

   Note: Use of the empty insert_column_list is considered poor programming practice. The interpretation of the empty insert_column_list is subject to change with time, as the database design changes. Programs which use an empty insert_column_list may cease functioning where a program supplying an insert_column_list would continue to operate correctly.

4. If the statement is an insert_statement_values, then

   a. If the insert_value_list is not present, then a list consisting of the sequence of column names in the insert_column_list is assumed.

   b. The insert_column_list and insert_values_list must conform, as described in Section 5.8.

5. If the statement is an insert_statement_query, then let C_1, C_2 ..., C_m be the columns appearing in an *SQL*_insert_column_list, and for each $1 \leq i \leq m$, let D_i be DBMS_TYPE(C_i) (see section 4.2). The select_parameters in the select_list of the query_specification shall not specify a named_phrase or a not null phrase; that is, the select_list shall have the form VE_1, VE_2, ..., VE_n, for value_expressions, VE_i. Then

- m = n, that is, the lists have the same length; and
- For each $1 \leq i \leq n$, VE_i shall conform to D_i (see section 3.5) and if DATACLASS(D_i) is character, then LENGTH(VE_i) shall not exceed LENGTH(D_i).

6. The following applies to update statements. Let

   C = v

   be a set_item within an update_statement. Let D be DOMAIN(C). Then

   a. If v is the null literal, then D shall not be defined as a non-null bearing domain.

   b. Otherwise, v is a value_expression. v shall conform to D (see section 3.5) and if DATACLASS(D) is character, LENGTH(v) shall not exceed LENGTH(D).

SQL Semantics

The text of an SQL statement corresponding to a SAMeDL statement within a procedure is described below.

1. The SAMeDL and SQL commit and rollback statements are textually identical.

2. The SAMeDL delete_statement is transformed into an *SQL*_delete_statement_searched by applying the transformation SQL_{SC} described in Section 5.11 to the search condition of the where clause, if present. The remainder of the statement is unchanged.

3. The SAMeDL insert_statement_query is transformed into an *SQL*_insert_statement by

   a. Applying the transformation SQL_{VE} defined in Section 5.10 to the value_expression in each select_parameter of the select_list in the query_specification.

   b. Removing any as keywords, if present, from the from_clause in the query_specification.

   c. Applying the transformation SQL_{SC} described in Section 5.11 to the search_conditions, if any, in the query_specification.

   The remainder of the statement is unchanged.

4. The SAMeDL insert_statement_values is transformed into an *SQL*_insert_statement by transforming the insert_values_list and insert_column_list as described in Section 5.8, and dropping the insert_from_clause, if present. The remainder of the statement is unchanged.

5. The SAMeDL update_statement is transformed into an *SQL*_update_statement_searched by applying the transformation SQL_{VE} to the value expressions in the set_items of the statement and by applying the transformation SQL_{SC} to the search condition, if present. The remainder of the statement is unchanged.

6. The SAMeDL select_statement is transformed into an *SQL*_select_statement by

a. Replacing the select_list with the *SQL*_select_list described in Section 5.7;

b. Inserting an SQL into clause with a target list as specified in Section 5.7, and removing the into_clause in the statement, if any;

c. Removing any as keywords, if present, from the from_clause.

d. Applying the transformation SQL_{SC} described in Section 5.11 to the search conditions, if any, in the where and having clauses.

The remainder of the statement is unchanged.

5.4 Cursor Declarations

```
cursor_declaration ::=    [ extended ] cursor Ada_identifier_1
                              [ input_parameter_list ]
                              for
                                  query
                                  [ SQL_order_by_clause ]
                              ;
                              [ is cursor_procedures
                          end [ Ada_identifier_2 ] ; ]

query ::= query_expression | extended_query_expression

query_expression ::= query_term |
                     query_expression union [ all ] query_term

query_term ::=    query_specification |
                  ( query_expression )

query_specification ::=    select [ distinct | all ] select_list
                               from_clause
                               [ where search_condition ]
                               [ SQL_group_by_clause ]
                               [ having search_condition ]

SQL_order_by_clause ::=    order by SQL_sort_specification { , SQL_sort_specification }

SQL_sort_specification ::=    Unsigned_integer_literal [ asc | desc ] |
                              column_reference [ asc | desc ]
```

1. *Ada*_identifier_1 is the *name* of the cursor. If present, *Ada*_identifier_2 shall equal *Ada*_identifier_1

2. No two procedures within a cursor_declaration shall have the same name.

3. A query may be an extended_query_expression only if the keyword **extended** appears in the cursor declaration. If the keyword **extended** appears in the cursor declaration, then the keyword **extended** shall appear in the declaration of the module in which the cursor is declared.

Ada Semantics

If a cursor named C is declared within an abstract module named M, then there exists within the Ada package M (see Section 5.1) a subpackage named C. That subpackage shall contain the

declarations of the procedures declared in the sequence cursor_procedures. (***Note:*** Some of those procedures may appear by assumption. See Section 5.5). The text of the procedure declarations is described in Section 5.5.

If there is no **union** operator in the query_expression in the cursor_declaration, then the names, types, and order of the components of any record type used as a row record formal parameter type in any fetch procedure for this cursor are determined from the select_list as specified for the select_statement in Sections 5.2 and 5.7. Otherwise, if **union** is present, the select_lists of all the query_expressions in the cursor_declaration shall have the same length. The name and type of the i^{th} component of the record type is determined by the set of select_parameters in the i^{th} location of the select_lists. Let there be m such select_lists and let the set of select_parameters appearing in the i^{th} location of these lists be denoted by

$$\{SP^j_i\} = \{VE^j_i\ [\ \textbf{named}\ Id^j_i\ ]\ [\ \textbf{not null}^j_i\ ]\ [\ \textbf{dblength}^j_i\ ]\ [\ \textbf{named}\ dbId^j_i\ ]\ \}\ 1 \le j \le m.$$

Then

1. These parameters have the same Ada type; that is, $AdaTYPE(SP^j_i) = AdaTYPE(SP^k_i)$ for all pairs $1 \le j, k \le m$ (see Section 5.7). The Ada type of the i^{th} parameter, $AdaTYPE_i$, is that type; in other words, $AdaTYPE_i = AdaTYPE(SP^j_i)$ for any $1 \le j \le m$. (***Note:*** This is equivalent to the restriction that $DOMAIN(VE^j_i)$ is the same domain, say $DOMAIN_i$, for all values of j (see Section 5.10) and that either (i) $DOMAIN_i$ is a not null only domain, or (ii).**not null** is specified for either all or none of the parameters).

2. For all pairs j, k such that a named_phrase appears in SP^j_i and SP^k_i, Id^j_i shall equal Id^k_i. Then that name, $AdaNAME_i$, satisfies $AdaNAME_i = Id^j_i$ for any such j. If there are no such pairs (that is, if a **named** phrase appears in none of the select_parameters), then $AdaNAME(VE^j_i)$ shall equal $AdaNAME(VE^k_i)$ for all pairs $1 \le j, k \le m$ and shall not equal NO_NAME (see Section 5.10). Then $AdaNAME_i = AdaNAME(VE^j_i)$ for any $j \le m$.

3. For all pairs j, k such that a **dblength** phrase appears in SP^j_i or SP^k_i, then a **dblength** phrase shall appear in both SP^j_i and SP^k_i. Furthermore, $DBLngNAME(SP^j_i)$ shall equal $DBLngNAME(SP^k_i)$. Then $DBLngNAME_i$ shall be that name. If the **dblength** phrase appears in no SP^j_i for any j, then $DBLngNAME_i$ is said to be null; otherwise, $DBLngNAME_i$ is undefined.

The type of the row record parameter is equivalent in the sense **[Ada]** 3.2.10 and 3.7.2, to a record type whose sequence of components is given by the sequence

$$\ldots\ COMP_{Ada}(SP_i)\ [\ DBleng_{Ada}(SP_i)\ ]$$

where $COMP_{Ada}(SP_i)$ is given by

$$AdaNAME(SP_i)\ :\ AdaTYPE(SP_i)$$

provided that $AdaNAME(SP_i)$ and $AdaTYPE(SP_i)$ are defined (see Section 5.7). The record component $COMP_{Ada}(SP_i)$ is otherwise undefined. The record component $DBLeng_{Ada}(SP_i)$ is given by

DBLngNAME(SP_i) : Ada_Indicator_Type

where Ada_Indicator_Type is the type SQL_Standard.Indicator_Type (see **[ESQL]** section 8.3.a), *provided* that DBLngNAME(SP_i) is defined; otherwise this component is not present.

SQL Semantics

A SAMeDL cursor_declaration is transformed into an *SQL*_cursor_declaration as follows.

1. The string "declare" is prepended to the cursor declaration.
2. The input_parameter_list and cursor_procedures are discarded, as is the keyword **cursor** and the **is ... end** bracket. The cursor name *Ada*_identifier_1 is transformed into SQL_{NAME}(*Ada*_identifier_1).
3. The string "cursor" is inserted immediately after the transformed cursor name, but before the keyword **for**.
4. The select_list is transformed into an *SQL*_select_list as described in Section 5.7.
5. Any **as** keywords present are removed from the from_clause.
6. The search conditions are transformed using the transform SQL_{SC} of Section 5.11.

The remainder of the declaration is unchanged.

Examples:

Shown below are two examples of cursor declarations: the first contains a simple cursor declaration, while the second contains a more complex declaration which exercises many of the features of the syntax. In both cases, the generated Ada code is shown.

The example below is a simple SAMeDL cursor declaration.

```
cursor Select_Suppliers
    for
        select Sno, Sname, Sstatus, City dblength
            from S
    ;
```

This declaration produces the following Ada code:

```
package Select_Suppliers is                          -- 5.4: Ada Semantics
    type Row_Type is record                          -- 5.5, #5 and #8
        Sno          : Sno_Domain_Type;              -- 5.5, #3 and #8
        Sname        : Sname_Domain_Type;
        Sstatus      : Sstatus_Domain_Type;
        City         : City_Domain_Type;
        City_Dblength : SQL_Standard.Indicator_Type;
    end record;

    procedure Open;                                  -- 5.5, #3
```

```
        procedure Fetch (                          -- 5.5, #5
            Row       : in out Row_Type;           -- 5.5, #8 and Ada Semantics #3
            Is_Found  : out boolean);              -- 5.5, #5 and Ada Semantics #6

        procedure Close;                           -- 5.5, #4

    end Select_Suppliers;
```

The following is an example of a more complex cursor declaration.

```
    cursor Supplier_Operations (
        Input_City named Supplier_City          : City_Domain not null;
        Adjustment named Status_Adjustment      : Sstatus_Domain not null)

    for
        select Sno named Supplier_Number,
               Sname named Supplier_Name,
               Sstatus + Adjustment named Adjusted_Status,
               City named Supplier_City
        from S
        where City = Input_City
    ;

    is
        procedure Open_Supplier_Operations is
            open Supplier_Operations;

        procedure Fetch_Supplier_Tuple is
            fetch Supplier_Operations
                into Supplier_Row_Record : new Supplier_Row_Record_Type
                status My_Map named Fetch_Status;

        procedure Close_Supplier_Operations is
            close;  -- optional 'cursor name' omitted

        procedure Update_Supplier_Status (
            Input_Status named Updated_Status  : Sstatus_Domain not null;
            Input_Adjustment named Adjustment  : Sstatus_Domain)
        is
            update S
                set Sstatus = Input_Status + Input_Adjustment
                where current of Supplier_Operations;

        procedure Delete_Supplier is
            delete from S;
            -- optional "where current of 'cursor name'" omitted

    end Supplier_Operations;
```

This declaration produces the following Ada code.

```
    package Supplier_Operations is                      -- 5.4, Ada Semantics
        type Supplier_Row_Record_Type is record         -- 5.5, Ada Semantics #5 and #8
            Supplier_Number   : Sno_Domain_Type;        -- 5.5, Ada Semantics #3 and #8
            Supplier_Name     : Sname_Domain_Type;
```

```
        Adjusted_Status     : Sstatus_Domain_Type;
        Supplier_City       : City_Domain_Type;
    end record;

    procedure Open_Supplier_Operations (
        Supplier_City       : in City_Domain_not_null;          -- 5.5, Ada Semantics
        Status_Adjustment   : in Sstatus_Domain_not_null);      --       #1, #3, Modes

    procedure Fetch_Supplier_Tuple (
        Supplier_Row_Record : in out Supplier_Row_Record_Type;  -- 5.5
        Fetch_Status        : out Operation_Status);            -- 5.5

    procedure Close_Supplier_Operations;

    procedure Update_Supplier_Status (
        Updated_Status      : in Sstatus_Domain_not_null;       -- 5.5, Ada Semantics #2
        Adjustment          : in Sstatus_Domain_Type);          -- 5.5, Ada Semantics

    procedure Delete_Supplier;

end Supplier_Operations;
```

5.5 Cursor Procedures

```
cursor_procedures ::= cursor_procedure { cursor_procedure }

cursor_procedure ::= [ extended ] procedure Ada_identifier_1
                         [ input_parameter_list ]
                     is
                         cursor_statement
                         [ status_clause ]
                     ;

cursor_statement ::= open_statement              |
                     fetch_statement             |
                     close_statement             |
                     cursor_update_statement     |
                     cursor_delete_statement     |
                     extended_cursor_statement

open_statement ::= open [ Ada_identifier ]

fetch_statement ::= fetch [ Ada_identifier t ] [ into_clause ]

close_statement ::= close [ Ada_identifier ]

cursor_update_statement ::= update table_name
                                set set_item { , set_item ]
                                [ where current of Ada_identifier ]

cursor_delete_statement ::= delete from table_name
                                [ where current of Ada_identifier ]
```

1. *Ada*_identifier_1 is the *name* of the procedure.

2. An input_parameter_list may only appear in conjunction with statements that take input parameters. In particular, such lists may not appear in conjunction with open, close, fetch and cursor delete statements. Of the cursor procedures, only a cursor_update_statement may take an input_parameter_list.

3. If no open_statement appears in a list of cursor_procedures, the declaration "**procedure** open **is open**;" is assumed.

4. If no close_statement appears in a list of cursor_procedures, the declaration "**procedure** close **is close**;" is assumed.

5. If no fetch_statement appears in a list of cursor_procedures, the declaration "**procedure** fetch **is fetch status** Standard_Map;" is assumed. See Section 4.1.8.

6. If *Ada_*identifier is present in an open, fetch, close, cursor_update or cursor_delete_statement, then it must be equal to the name of the cursor within which the procedure declaration appears. The meaning of a cursor statement is not affected by the presence or absence of these identifiers.

7. The restrictions which apply to the set_items of a non-cursor update_statement (see Section 5.3), also apply to the set items of a cursor_update_statement.

8. If no into_clause appears within a fetch_statement, then the following clause is assumed:

   into Row : **new** Row_Type

   If an into_clause which does not contain a record_id appears in a fetch_statement, the record_id

   : **new** Row_Type

   is assumed. See Section 5.9.

9. A cursor_statement may be an extended_cursor_statement only if the keyword **extended** appears in the cursor_procedure declaration. If the keyword **extended** appears in the cursor_procedure declaration, then the keyword **extended** shall appear within the declaration of the cursor in which the cursor_procedure is declared.

Ada Semantics

Each procedure declaration P which appears in or is assumed to appear in a cursor_procedures list shall be assigned an Ada procedure declaration P_{Ada} which satisfies the following constraints.

- If P is declared within the declaration of a cursor named C, then P_{Ada} shall be declared within the specification of an Ada subpackage named C.
- The simple name of P_{Ada} is the name of P.

The parameter profiles (Ada formal parts) of the Ada procedures depend in part on the statement within the procedure, as follows:

1. For open_statements: Let INP_1, INP_2, ..., INP_k $k \geq 0$ be the list of input parameters in the input_parameter_list of the cursor_declaration within which the procedure appears. Then $PARM_{Ada}(INP_i)$, the i^{th} parameter of the *Ada*_formal_part, is of the form

   AdaNAME(INP_i) : in AdaTYPE(INP_i)

   for $1 \leq i \leq k$ (see Section 5.6).

2. For cursor_update_statements: Let INP_1, INP_2, ..., INP_k $k>0$ be the list of input parameters in the input_parameter_list of the statement. Then $PARM_{Ada}(INP_i)$, the i^{th} parameter of the *Ada*_formal_part, is of the form

   AdaNAME(INP_i) : in AdaTYPE(INP_i)

   for $1 \leq i \leq k$ (see Section 5.6).

3. For fetch_statements: The first parameter is a row record parameter of mode **in out**. The names, order and types of the components of the type of this parameter are described in Sections 5.2 and 5.4. Let *IC* be the into_clause of the fetch_statement. Then the name of the row record formal parameter is $PARM_{Row}(IC)$, and the name of the type of that parameter is $TYPE_{Row}(IC)$. See Section 5.9. If IC contains the keyword **new**, then the declarative region containing the declaration of P_{Ada} shall contain the declaration of $TYPE_{Row}(IC)$.

4. For close and cursor_delete_statements: There are no parameters to these procedures (except possibly for the status parameter, see below).

5. For all statement types: if a status_clause referencing a status map that contains a **uses** appears in the procedure declaration, then the final parameter is a status parameter of mode **out**. For the name and type of this parameter see Sections 4.1.8 and 5.13.

SQL Semantics

Each procedure P which appears in or is assumed to appear in a cursor_procedures list shall be assigned an SQL procedure P_{SQL} within the SQL module for the abstract module within which the cursor_procedures list appears. P_{SQL} has three parts:

1. An *SQL*_procedure_name. This is implementation defined.

2. A list of *SQL*_parameter_declarations. An SQLCODE parameter is declared for every SQL procedure. Other parameters depend on the type of the statement within the procedure P.

   a. If the statement is an open_statement, then the SQL parameters derived from the input_parameter_list of the cursor_declaration as described in Section 5.6 appear in the parameter declarations of P_{SQL}.

   b. If the statement is a cursor_update_statement, then the SQL parameters derived from the input_parameter_list of the cursor_update_statement as described in Section 5.6 appear in the parameter declarations of P_{SQL}.

c. If the statement is a fetch_statement, then the SQL parameters determined by the select_list of the cursor_declaration as described in Section 5.7 appear in the parameter declarations of P_{SQL}.

The order of the parameters within the list is implementation defined.

3. An *SQL*_SQL_statement. This is derived from the statement in the procedure declaration, as follows.

   a. If the statement is an open_statement, then the *SQL*_open_statement is "**open** $SQL_{NAME}(C)$", where C is the cursor name.

   b. If the statement is a close_statement, then the SQL close statement is "**close** $SQL_{NAME}(C)$", where C is the cursor name.

   c. If the statement is the cursor_delete_statement

      delete from id [**where current of** C]

      then the *SQL*_delete_statement_positioned is identical, up to the addition of the where phrase: "**where current of** $SQL_{NAME}(C)$", replacing the where phrase of the cursor_delete_statement, if present.

   d. If the statement is the cursor_update_statement

      update id
      set *set_items*
      [**where current of** C]

      then the *SQL*_update_statement_positioned is formed by applying the transformation SQL_{VE} defined in Section 5.10 to the value expressions in the set_items of the statement and appending or replacing the where phrase so as to read "**where current of** $SQL_{NAME}(C)$".

   e. If the statement is a fetch_statement, then the *SQL*_fetch_statement is "**fetch** $SQL_{NAME}(C)$ **into** *target_list*" where C is the cursor name and *target list* is described in Section 5.7.

Interface Semantics

A call to the Ada procedure P_{Ada} shall have effects which can not be distinguished from the following.

1. The procedure P_{SQL} is executed in an environment in which the values of parameters $PARM_{SQL}(INP)$ and $INDIC_{SQL}(INP)$ (see Section 5.6) are set from the value of $PARM_{Ada}(INP)$ (see Ada semantics above) for every input parameter, INP, in either the input_parameter_list of the cursor_declaration, for open procedures, or the input_parameter_list of the procedure itself, for update procedures.

2. Standard post processing, as described in section 3.6 is performed.

3. If the value of the SQLCODE parameter is zero or an implementation defined value which permits the transmission of data (and which is handled by an sqlcode_assignment, see section 3.6),, and the statement within the procedure is a

fetch_statement, then the value of the row record components $COMP_{Ada}(SP_i)$ and $DBLeng_{Ada}(SP_i)$, are set from the values of the actual parameters associated with the SQL formal parameters $PARM_{SQL}(SP_i)$ and $INDIC_{SQL}(SP_i)$ (see Section 5.7).

5.6 Input Parameter Lists

Input parameter lists declare the input parameters of the procedure or cursor declaration in which they appear. The list consists of input parameter declarations which are separated with semi-colons, in the manner of Ada formal parameter declarations.

Each parameter declaration of a procedure P is represented as an *Ada*_parameter_specification within the *Ada*_formal_part of the procedure P_{Ada}; each parameter declaration within a cursor declaration is represented as an *Ada*_parameter_specification within the *Ada*_formal_part of the Ada open procedure. The parameter is also represented as either one or two *SQL*_parameter_declarations within the *SQL*_procedure P_{SQL}. The second SQL parameter declaration, if present, declares the indicator variable for the parameter ([SQL] 4.10.2).

The order of parameter specification within the *Ada*_formal_part is given by the order within the input_parameter_list. The order of the *SQL*_parameter_declarations within the list of declarations in the SQL procedure is implementation defined.

input_parameter_list ::= (parameter { ; parameter })

input_parameter ::= *Ada*_identifier_1 [named_phrase] :
[**in**] [**out**] domain_reference [**not null**]

Ada Semantics

Let INP be a parameter the textual representation of which is given by

id_1 [**named** id_2] : [**in**] [**out**] [id_3.] id_4 [**not null**]

Then id_1 is the *name* of the parameter.

The domain associated with INP, denoted DOMAIN(INP), is the domain referenced by [id_3.]id_4. Let DOMAIN(INP) = D. Then

- LENGTH(INP) = LENGTH(D)
- SCALE(INP) = SCALE(D)
- DATACLASS(INP) = DATACLASS(D)

The functions AdaNAME and AdaTYPE are defined on parameters as follows:

1. If id_2 is present in the definition of INP, then AdaNAME(INP) = id_2 otherwise, AdaNAME(INP) = id_1. For no two parameters, INP_1 and INP_2, in an input parameter list shall $AdaNAME(INP_1) = AdaNAME(INP_2)$.

2. AdaTYPE(INP) shall be the name of a type within the domain identified by the domain_reference [id_3.]id_4. If **not null** appears within the textual representation of INP, or the domain identified by the domain_reference does not null only, then AdaTYPE(INP) shall be the name of the not null-bearing type within the identified

domain; otherwise it shall be the name of the null-bearing type within that domain (see Section 4.1.3).

The optional **out** may occur only in a parameter that is associated with a procedure or cursor that is extended. The optional **in**, however, may be included in any parameter declaration.

Given INP as defined above, define MODE(INP) to be

- *in*, if INP either contains (1) the optional **in**, but not the optional **out**, or (2) neither **in** nor **out**.
- *out*, if INP contains **out** but not **in**.
- *in out*, if INP contains both **in** and **out**.

Then the generated parameter, $PARM_{Ada}$(INP), in the *Ada*_formal_part is of the form

AdaNAME(INP) : MODE(INP) AdaTYPE(INP);

SQL Semantics

Let INP be as given above and let *D* be the domain referenced by [id_3.]id_4.. The *SQL*_parameter_declaration $PARM_{SQL}$(INP) is declared by the following

: SQL_{NAME}(id_1) DBMS_TYPE(D)

where DBMS_TYPE(D) is as given in Section 4.1.3. If **not null** does not appear within the textual representation of INP, and [id_3.]id_4 does not identify a not null only domain, then the parameter $INDIC_{SQL}$(INP) is defined and has a textual representation given by the *SQL*_parameter_declaration

: $INDIC_{NAME}$(INP) *indicator_type*

where *indicator_type* is the implementation defined type of indicator parameters ([SQL] 5.6.2). The name $INDIC_{NAME}$(INP) shall not appear as the name of any other parameter of the enclosing procedure.

5.7 Select Parameter Lists

Select parameter lists serve to inform the DBMS what data are to be retrieved by a select or fetch statement. They also specify the names and types of the components of a record type - the so called *row record type* - which appears as the type of a formal parameter of Ada procedure declarations for select and fetch statements. Further they specify the column names of viewed tables (see section 4.2.2).

select_list ::= * | select_parameter { , select_parameter }

select_parameter ::= value_expression [named_phrase] [**not null**]
[**dblength** [named_phrase]]

1. The select list star ("*") is equivalent to a sequence of select parameters described as follows: Let T_1, T_2, ..., T_k be the list of exposed table names in the table expression **from** clause for the query specification in which the select list appears (see [SQL] Section 5.25).

Let U_i, for $1 \le i \le k$ be defined as $S_i\_V_i$ if T_i is of the form $S_i.Vi$ (i.e., S_i is a schema_name, and V_i is a table name); otherwise, U_i is T_i. In other words, U_i is T_i with every "." replaced by an underscore "_". Let $A_{i,1}, A_{i,2}, \ldots, A_{i,m_i}$, be the names of the columns of the table named T_i. Then the select list is given by the sequence $T_1.A_{1,1}$ **named** $U_1\_A_{1,1}$, $T_1.A_{1,2}$ **named** $U_1\_A_{1,2}$, ..., $T_i.A_{i,j}$ **named** $U_i\_A_{i,j}$, ..., $T_k.A_{k,m}$ **named** $U_k\_A_{k,m}$ That is, the columns are listed in the order in which they were defined (see Section 4.2) within the order in which the tables were named in the **from** clause.

Note: This definition differs from that given in **[SQL]** Section 5.25 (4) in specifying that the column references are qualified by table name or correlation name. The record type being described must have well defined component names.

Note: Use of "*" as a select list in an abstract module is considered poor programming practice. The interpretation of "*" is subject to change with time, as the database design changes. Programs which use a "*" may cease functioning where a program using a named select list would continue to operate correctly.

In the following discussion, assume that a select list "*" has been replaced by its equivalent list, as described above.

2. If the keyword **dblength** is present, then value_expression shall have the data class **character**.

3. Let VE be the value_expression appearing in a select_parameter. DOMAIN(VE) shall not be NO_DOMAIN and VE shall conform to DOMAIN(VE).

Ada Semantics

Let SP be a select parameter written as

VE [**named** *id_1*] [**not null**] [**dblength** [**named** *id_2*]]

SP is assigned the Ada type name AdaTYPE(SP), the Ada name AdaNAME(SP) and the dblength name DBLngNAME(SP) as follows:

- Let DOMAIN(VE) = *D* (see Section 5.10) where $D \neq$ NO_DOMAIN. If **not null** appears in SP or D is a not null only domain, then AdaTYPE(SP) is the name of the not null-bearing type name within the domain *D*; else AdaTYPE(SP) is the name of the null-bearing type within the domain *D*.

- If DOMAIN(VE) = NO_DOMAIN then AdaTYPE(SP) is undefined.

- If *id_1* appears in SP, then AdaNAME(SP) = *id_1*; else AdaNAME(SP) = AdaNAME(VE) (see Section 5.10).

- If the **dblength** phrase appears in SP, then

  -- If *id_2* is present then DBLngNAME(SP) = *id_2*

  -- else, DBLngNAME(SP) = AdaNAME(SP)_DbLength

  Otherwise, DBLngNAME(SP) is undefined.

- DBLngNAME(SP) and AdaNAME(SP) shall not appear as either DBLngNAME(SP_i) or as AdaNAME(SP_i) for any other select_parameter SP_i within the select_list that contains SP.

SQL Semantics

From a select_list, three SQL fragments must be derived:

1. An *SQL*_select_list, within the select statement or cursor declaration
2. A list of *SQL*_parameter_declarations.
3. An *SQL*_target_list, within a select statement or fetch statement.

An *SQL*_select_list is derived from a select_list as follows:

- The select_list * becomes the *SQL*_select_list *.
- Otherwise, suppose $SP_1, SP_2, \ldots, SP_n$ is a select_list, where SP_i is given by:

  VE_i [**named** *id_1*$_i$] [**not null**]$_i$ [**dblength**$_i$ [**named** *id_2*$_i$]]

  The *SQL*_select_list, $SP'_1, SP'_2, \ldots, SP'_n$ is formed by setting SP'_i to $SQL_{VE}(VE_i)$.

For the purpose of defining the *SQL*_parameter_declarations and target list generated from a select_list, let $SP_1, SP_2, \ldots, SP_n$ be the select_list supplied or the select_list that replaced the select_list * as described above. Let each SP_i be be as given above. Then

- There are two SQL parameters associated with each select_parameter, SP_i. They are $PARM_{SQL}(SP_i)$ and $INDIC_{SQL}(SP_i)$, where the *SQL*_parameter_declaration declaring $PARM_{SQL}(SP_i)$ is

  : $SQL_{NAME}(SP_i)$ DBMS_TYPE(DOMAIN(VE_i))

  and the *SQL*_parameter_declaration declaring $INDIC_{SQL}(SP_i)$ is

  : $INDIC_{SQL}(SP_i)$ *indicator_type*

  where $SQL_{NAME}(SP_i)$ and $INDIC_{SQL}(SP_i)$ are *SQL*_identifiers not appearing else where.

- The target list generated from a select_list is a comma-separated list of *SQL*_target_specifications ([SQL], section 5.6). The i^{th} *SQL*_target_specification in the SQL target list is

  SQL_{NAME} (SP_i) INDICATOR : $INDIC_{NAME}(SP_i)$

  Note: All derived target specifications contain indicator parameters, irrespective of the presence or absence of a **not null** phrase in the select parameter declaration.

5.8 Value Lists And Column Lists

insert_column_list ::= insert_column_specification { , insert_column_specification }

insert_column_specification ::= column_name [named_phrase] [**not null**]

insert_value_list ::= insert_value { , insert_value }

```
insert_value ::=  null                        |
                  constant_reference          |
                  literal                     |
                  column_name                 |
                  domain_parameter_reference
```

Each column_name within a insert_column_list shall specify the name of a column within the table into which insertions are to be made by the enclosing insert_statement_values. (See Section 5.3. See also **[SQL]**, 8.7(3).)

Let *C* be the insert_column_specification

Col [**named** *id*] [**not null**] .

Then AdaNAME(C) is defined to be *id*, if *id* is present; otherwise it is *Col*. Let DOMAIN(C) = DOMAIN(Col) = D be the domain assigned to the column named Col. If **not null** appears in C, or D is a not null only domain, then AdaTYPE(C) is the name of the not null-bearing type within the domain D; otherwise, AdaTYPE(C) is the null-bearing type within the domain D.

Let CL be the insert_column_list C_1, ..., C_m ; let IL be the insert_value_list V_1, ..., V_n. CL and IL are said to conform if:

1. m=n, that is, the length of the two lists is the same;
2. For each $1 \le i \le m$, if V_i is
   a. The literal **null**, then DOMAIN(C_i) shall not be a not null only domain..
   b. A literal or reference to either a constant or a domain parameter, then V_i shall conform to DOMAIN(C_i) (see section 3.5) and if DATACLASS(DOMAIN(C_i)) is character, then LENGTH(V_i) shall not exceed LENGTH(DOMAIN(C_i)).
   c. A column_name, then V_i shall be identical to the column_name in C_i.

Ada Semantics

The insert_column_list and insert_value_list of an insert_statement_values together define the components of an Ada record type declaration. The names, types and order of those components are defined in Section 5.2 on the basis of the functions AdaNAME and AdaTYPE described above. For the name of the record type and its place of declaration, see Section 5.9.

Note: If the insert_values_list contains no column_names, then the Ada procedure corresponding to the procedure containing the insert_statement_values statement of which these lists form a part does not have a row record parameter. See Section 5.2.

SQL Semantics

A set of SQL parameter declarations is defined from the pair of insert_column_list and insert_value_list. So again let C_1, ..., C_k be the subsequence of the insert_column_list such that the insert_value_list item corresponding to each C_i is a column_name (and therefore neither a literal nor a constant reference nor a domain parameter reference). Further, let C_i be represented by the text string

COL_i [**named** *id*$_i$] [**not null**$_i$] .

Then the SQL parameter declarations $PARM_{SQL}(C_i)$ for $1 \le i \le k$ given by

$SQL_{NAME}(Col_i)$ DBMS_TYPE(DOMAIN(Col_i))

appear in the list of SQL parameter declarations, where

1. $SQL_{NAME}(Col_i)$ is an implementation-defined *SQL*_identifier which appears nowhere else.

2. DBMS_TYPE(DOMAIN(Col_i)) is as defined in section 4.1.3.

If **not null** does *not* appear in C_i *and* the domain DOMAIN(Col_i) is *not* not null only, then the parameter $INDIC_{NAME}(C_i)$ is defined and the parameter declaration

$INDIC_{NAME}(C_i)$ *indicator_type*

also appears in the list of SQL parameter declarations, where

1. $INDIC_{NAME}(C_i)$ is an implementation-defined *SQL*_identifier that appears nowhere else.

2. *indicator_type* is the implementation-defined type of indicator parameters (**[SQL]** 5.6.2).

An insert_column_list and insert_value_list pair are transformed into an *SQL*_insert_column_list and *SQL*_insert_value list pair as follows:

1. An insert_column_list is transformed into an *SQL*_insert_column_list by the removal of all named_phrase and **not null** phrases that appear in it.

   Note: This implies that the empty insert_column_list is transformed into the empty *SQL*_insert_column_list.

2. An insert_value_list is transformed into an *SQL*_insert_value list by replacing each list element as follows:

   a. a literal (including the literal **null** but excluding any enumeration literal) is replaced by itself; i.e., it is unchanged;

   b. a constant_reference, enumeration_literal, or domain_parameter_reference *k* is replaced by a textual representation of its database value $SQL_{VE}(k)$ (see Section 5.10).

   c. a column_name Col_i is replaced by

$SQL_{NAME}(Col_i)$ [INDICATOR : $INDIC_{NAME}(C_i)$]

where the INDICATOR phrase appears whenever the indicator parameter, $INDIC_{SQL}(C_i)$, is defined (see above).

enclosing the resulting list in parentheses and preceding it with the keyword **values**.

5.9 Into_Clause And Insert_From_Clause

An into_clause is used within a select_statement or a fetch_statement, and an insert_from_clause is used within an insert_statement_values, to explicitly name the row record parameter of those statements and/or the type of that parameter.

```
into_clause ::= into ß

insert_from_clause ::= from into_from_body

into_from_body ::=   Ada_identifier_1 : record_id   |
                     Ada_identifier_1               |
                     : record_id

record_id ::= new Ada_identifier_2   |
              record_reference
```

Ada Semantics

Define the string $PARM_{Row}(IC)$ as follows, where IC is an into_clause or insert_from_clause.

1. If *Ada*_identifier_1 appears in IC, then $PARM_{Row}(IC)$ = *Ada*_identifier_1.

2. Otherwise, if the record_id takes the form of a record_reference referencing the record declaration R, then $PARM_{Row}(IC)$ = AdaNAME(R) (see Section 4.1.3).

3. Otherwise, $PARM_{Row}(IC)$ = *Row*.

Define $TYPE_{Row}(IC)$ as follows:

1. If record_id has the form "**new** *Ada*_identifier_2", then $TYPE_{Row}(IC)$ = *Ada*_identifier_2.

2. Otherwise, $TYPE_{Row}(IC)$ is the record type referenced by the record_reference.

Note: The assumptions made about into_clause and insert_from_clause in sections 5.3 and 5.5 are sufficient to ensure that every such clause contains a record_id, possibly by assumption. Therefore, the case of a missing record_id need not be considered in the definition of $TYPE_{Row}(IC)$ If the record_id is a record_reference, then the names, types, and order of the components of the record type declaration that would have been generated had the record_id been "**new** *Ada*_identifier" (see sections 5.2, 5.4, and 5.7).

Examples:

The following is a set of examples which illustrate various uses of **into** and **from** clauses. It is assumed that each of these procedures is declared within an abstract module, and that any enumeration, record, and status map declarations used are visible at the point at which each procedure is declared. In addition, it is assumed that the abstract modules in which these procedures are declared have direct visibility to the contents of the *Parts_Supplier_Database* schema module shown in Section 4.2.2.

The two examples below illustrate the use of a previously declared record object in the **into** clauses of select statements. These examples illustrate a possible scenario where an SQL module contains two select statements for the same object, namely a part. The first select statement below exists in a cursor declaration because it has the potential to return more than one record. The second select statement exists in a procedure, because it can return at most one record from the table. Since both select statements retrieve the same type of object from the database, they may share a row record. The row record contains the definition of the part abstraction. To share a record object, declare the record first, and then reference it in the **into** clauses of both select statements.

```
cursor Parts_By_City (
     Input_City named Part_Location : City_Domain not null)
for
     select  Pno named Part_Number not null,
             Pname named Part_Name,
             Color,
             Weight * 16 named Weight_In_Ounces,
             City
     into Parts_By_City_Row : Parts_Row_Record_Type
     from P
     where City = Input_City
;

procedure Parts_By_Number (
     Input_Pno named Part_Number : Pno_Domain not null)
is
     select  Pno named Part_Number not null,
             Pname named Part_Name,
             Color,
             Weight * 16 named Weight_In_Ounces,
             City
     into Parts_By_Number_Row : Parts_Row_Record_Type
     from P
     where Pno = Input_Pno
     status Operation_Map named Parts_By_Number_Status
;
```

The above declarations produce the following Ada declarations in the abstract interface.

```
package Parts_By_City is
     procedure Open (Part_Location : in City_Domain_not_null);

     procedure Fetch (
        Parts_By_City_Row     : in out Parts_Row_Record_Type;
        Is_Found              : out boolean);

     procedure Close;
```

```
end Parts_By_City;

procedure Parts_By_Number (
      Part_Number              : in Pno_Domain_not_null;
      Parts_By_Number_Row      : in out Parts_Row_Record_Type;
      Parts_By_Number_Status   : out Operation_Status);
```

The select procedure below illustrates the use of an **into** clause to specify the parameters, types, and names of the generated row record parameter.

```
procedure Part_Name_By_Number (
      Input_Pno named Part_Number : Pno_Domain not null)
is
      select Pname named Part_Name
      into Part_Name_By_Number_Row : new Part_Name_Row_Record_Type
      from P
      where Pno = Input_Pno
      status Operation_Map named Parts_By_Number_Status
;
```

The above declaration produces the following Ada record type and procedure declarations at the in the abstract interface.

```
type Part_Name_Row_Record_Type is record
      Part_Name : Pname_Domain_Type;
end record;

procedure Part_Name_By_Number (
      Part_Number              : in Pno_Domain_not_null;
      Part_Name_By_Number_Row  : in out Part_Name_Row_Record_Type;
      Parts_By_Number_Status   : out Operation_Status);
```

The example declaration below uses the default **from** clause, which produces a record declaration in the abstract interface.

```
procedure Add_To_Suppliers is
      Insert into S (Sno, Sname, Sstatus, City)
      values
      status Operation_Map named Insert_Status
;
```

The above procedure declaration produces the following Ada code in the abstract interface.

```
type Add_To_Suppliers_Row_Type is record
      Sno         : Sno_Domain_Type;
      Sname       : Sname_Domain_Type;
      Sstatus     : Sstatus_Domain_Type;
      City        : City_Domain_Type;
end record;

procedure Add_To_Suppliers (
      Row               : in Add_To_Suppliers_Row_Type;
```

```
        Insert_Status   : out Operation_Status);
```

This last example illustrates an insert values procedure declaration where all of the values are literals, meaning that no row record parameter is needed for the procedure declaration at the interface.

```
procedure Add_To_Parts is
    insert into P (Pno, Pname, Color, Weight, City)
    values ('P02367', 'RIGHT FENDER: TOYOTA', LT_RED, 25, 'PITTSBURGH')
    status Operation_Map named Insert_Status
;
```

The above declaration produces the following Ada procedure declaration in the abstract interface.

```
procedure Add_To_Parts (Insert_Status : out Operation_Status);
```

5.10 Value Expressions

The concrete syntax of SAMeDL value expressions differs from the concrete syntax of SQL value expressions in the following ways:

1. An operand of a SAMeDL value expression may be a reference to a constant, domain parameter, or enumeration literal defined either in a definitional module or in the enclosing abstract module.

2. Value expressions are strongly typed; therefore, a domain conversion operation must be introduced.

```
value_expression ::= term                          |
                     value_expression + term       |
                     value_expression - term

term ::=  factor          |
          term * factor   |
          term / factor

factor ::= [ + | - ] primary

primary ::=  literal                          |
             constant_reference               |
             domain_parameter_reference       |
             column_reference                 |
             input_reference                  |
             set_function_specification       |
             domain_conversion                |
             ( value_expression )

set_function_specification ::= count ( * )          |
                               distinct_set_function |
                               all_set_function

distinct_set_function ::= [ avg | max | min | sum | count ] ( distinct column_reference )
```

all_set_function ::= [**avg** | **max** | **min** | **sum**] ([**all**] value_expression)

domain_conversion ::= domain_reference (value_expression)

Five mappings are defined on value_expressions: **AdaNAME, DOMAIN, DATACLASS, LENGTH,** and **SCALE.**

The mapping AdaNAME calculates the default names of row record components when value expressions appear in select parameter lists. The range of AdaNAME is augmented by the special value **NO_NAME**, the value of AdaNAME for literals and non-simple names.

The mapping DOMAIN assigns a domain to each well-formed value expression. (***Note:*** If DOMAIN is not defined on a value expression, then the value expression is not well-formed). The class of domains is augmented by the special value **NO_DOMAIN**, the domain of literals and universal constants.

The mapping DATACLASS assigns a data class to each well-formed value expression. If the expression is a literal or universal constant (or composed solely of literals and universal constants), that is if DOMAIN(VE)= NO_DOMAIN, then the mapping returns the data class of the literal or universal constant (see section 2.4).

The mapping LENGTH returns the number of characters in a character string. LENGTH returns the special value **NO_LENGTH** on operands whose data class is not **character**.

The mapping SCALE returns the scale of the result of a numeric expression as determined by SQL. SCALE returns the special value **NO_SCALE** on operands whose data class is not numeric.

The mappings AdaNAME, DOMAIN, DATACLASS, LENGTH and SCALE are defined recursively as follows:

Base Cases:

1. **Literals.** Let L be a literal. Then AdaNAME(L) = NO_NAME, DOMAIN(L) = NO_DOMAIN. DATACLASS(L), LENGTH(L), and SCALE(L) are as defined in Section 2.4.

2. **References.** Let F be an input_reference, a constant_reference, a domain_parameter_reference, or a column_reference; let G be the object to which F makes reference. Then

   - AdaNAME(F) = F if F is the simple name of G; otherwise, AdaNAME(F) = NO_NAME.
   - DOMAIN(F) = DOMAIN(G),
   - DATACLASS(F) = DATACLASS(G),
   - LENGTH(F) = LENGTH(G), and
   - SCALE(F) = SCALE(G).

See sections 5.6, 4.1.4, 4.1.3, and 4.2.1.

Recursive Cases:

1. **Set functions.** Let SF be a set function and let VE be a value expression.
   - AdaNAME(SF(VE)) = NO_NAME.
   - If SF is MIN or MAX, then
     - DOMAIN(SF(VE)) = DOMAIN(VE),
     - DATACLASS(SF(VE)) = DATACLASS(VE)
     - LENGTH(SF(VE)) = LENGTH(VE),
     - SCALE(SF(VE)) = SCALE(VE)
   - If SF is COUNT, then
     - DOMAIN(COUNT(VE)) = DOMAIN(COUNT(*)) = NO_DOMAIN
     - DATACLASS(COUNT(VE)) = DATACLASS(COUNT(*)) = **Integer**,
     - LENGTH(COUNT(VE)) = LENGTH(COUNT(*)) = NO_LENGTH
     - SCALE(COUNT(VE)) = SCALE(COUNT(*)) = 0 (see Section 2.4).
   - If SF is SUM, then
     - DOMAIN(SUM(VE)) = NO_DOMAIN
     - DATACLASS(SUM(VE)) = DATACLASS(VE) and shall be a numeric data class
     - LENGTH(SUM(VE)) = NO_LENGTH
     - SCALE(SUM(VE)) = SCALE(VE)
   - If SF is AVG, then
     - DOMAIN(AVG(VE)) = NO_DOMAIN
     - LENGTH(AVG(VE)) = NO_LENGTH
     - DATACLASS(VE) shall be numeric and DATACLASS(AVG(VE)) and SCALE(AVG(VE)) are implementation defined.
2. **Domain Conversions.** Let D be a domain reference, VE a value expression. Then
   - AdaNAME(D(VE)) = NO_NAME.
   - DOMAIN(D(VE)) = D *provided*

a. DATACLASS(D) and DATACLASS(VE) are both numeric. In this case SCALE(D(VE)) = SCALE(VE), and if SCALE(D) < SCALE(VE) then a warning message must be generated which will state, in effect, that the loss of scale implied by this conversion will not occur in the query execution. The warning message need not be generated if the value expression is in an assignment context (see section 3.5). LENGTH(D(VE)) = NO_LENGTH in this case.

b. DATACLASS(D) and DATACLASS(VE) are both character. In this case, LENGTH(D(VE)) = LENGTH(VE), and if LENGTH(D) < LENGTH(VE) then a warning message must be generated which will state, in effect, that the loss of length implied by this conversion will not occur in the query execution. The warning message need not be generated if the value expression is in an assignment context (see section 3.5). SCALE(D(VE)) = NO_SCALE in this case.

c. DATACLASS(D) and DATACLASS(VE) are both enumeration, *provided* that

   i. if DOMAIN(VE) ≠ NO_DOMAIN, then DOMAIN(VE) = D;

   ii. if DOMAIN(VE) = NO_DOMAIN, then the value of VE is an enumeration literal in the domain D (***Note:*** Thus domain conversion may play the role played by type qualification in Ada, **[Ada]** 4.7).

   LENGTH(D(VE)) = NO_LENGTH and SCALE(D(VE)) = NO_SCALE in this case.

-- DATACLASS(D(VE)) = DATACLASS(VE).

Note: These rules imply that the equalities

-- DATACLASS(DOMAIN(VE)) = DATACLASS(VE)

-- LENGTH(DOMAIN(VE)) = LENGTH(VE)

-- SCALE(DOMAIN(VE)) = SCALE(VE)

do *not* necessarily hold.

3. **Arithmetic Operators**. Let VE_1, VE_2 be value expressions. Let

   $DOMAIN(VE_1) = D_1$;
   $DOMAIN(VE_2) = D_2$;
   $DATACLASS(VE_1) = T_1$;
   $DATACLASS(VE_2) = T_2$;
   $SCALE(VE_1) = S_1$;
   $SCALE(VE_2) = S_2$;

   Then T_1 and T_2 shall be numeric classes and

   a. For unary operators (+, -)

- -- AdaNAME([+|-]VE_1) = NO_NAME.
- -- LENGTH([+|-]VE_1) = NO_LENGTH.
- -- DOMAIN([+|-]VE_1) = D_1.
- -- DATACLASS([+|-]VE_1) = T_1.
- -- SCALE([+|-]VE_1) = S_1.

b. Let *op* be any binary arithmetic operator. Then AdaNAME(VE_1 op VE_2) = NO_NAME. LENGTH(VE_1 op VE_2) = NO_LENGTH.

c. DATACLASS(VE_1 op VE_2) = max(T_1, T_2) where **float > fixed > integer.**

d. Recall that the DOMAIN mapping is defined for a value expression just in case that value expression is legal. The value expression VE_1 op VE_2 is a *legal* value expression if:

- -- $D_1 \neq$ NO_DOMAIN and $D_2 \neq$ NO_DOMAIN and either
  - -- $T_1 = T_2 =$ **fixed** and op is either multiplication or division; or
  - -- $D_1 = D_2$
- -- or D_1 = NO_DOMAIN or D_2 = NO_DOMAIN, and
  - -- $T_1 = T_2 =$ **integer**, or else
  - -- $T_1 \neq$ **integer** and then $T_2 \neq$ **integer**, or else
  - -- $T_1 =$ **fixed** and D_2 = NO_DOMAIN and op is either multiplication or division, or else
  - -- $T_2 =$ **fixed** and D_1 = NO_DOMAIN and op is multiplication
- -- otherwise, VE_1 op VE_2 is not a legal value expression.

e. if VE_1 op VE_2 is a legal value expression, then DOMAIN(VE_1 op VE_2) =

- -- NO_DOMAIN *provided* that either
  - -- $D_1 = D_2$ = NO_DOMAIN
  - -- or $D_1 \neq$ NO_DOMAIN and $D_2 \neq$ NO_DOMAIN and $T_1 = T_2 =$ **fixed** and op is either multiplication or division.
- -- D_1 *provided* that $D_1 \neq$ NO_DOMAIN
- -- D_2 *otherwise*

f. SCALE(VE_1 op VE_2) is given by

- -- if op is an additive operator ([+|-]), then the larger of S_1 and S_2

-- if op is multiplication, then the sum of S_1 and S_2

-- if op is division, then it is implementation defined.

Note: The following are consequences of the definitions above.

-- AdaNAME(VE) has a value other than NO_NAME only in the case where VE is a simple identifier.

-- The product and quotient of any two **fixed** quantities is always defined as a **fixed** quantity with no domain, much like the Ada <universal_fixed>. However, whereas in Ada no operations other than conversion are defined for such quantities, they may be used anywhere that a literal with **fixed** data class may be used.

-- The result of a COUNT set function is treated as though it were an integer literal (see **[SQL]** 5.8).

-- The result of a SUM set function on a value expression VE is treated as though it were a literal of the data class DATACLASS(VE) (see **[SQL]** 5.8).

-- The result of a AVG set function is treated as though it were a literal of an implementation defined data class and scale (see **[SQL]** 5.8).

SQL Semantics

The SQL value expression derived from a SAMeDL value expression VE is formed by removing all domain conversions, replacing all constants and domain parameters with their values and all enumeration literals with their database representations (see section 4.3).

Let SQL_{VE} represent the function transforming SAMeDL value_expressions into *SQL*_value_expressions. Let VE be a SAMeDL value_expression. $SQL_{VE}(VE)$ is given recursively as follows:

1. If VE contains no operators, then

   a. If VE is a column reference or a database literal, then $SQL_{VE}(VE)$ is VE.

   b. If VE is an enumeration literal of domain D, and D assigns expression E to that enumeration literal (see rule 12 of section 4.1.3), then $SQL_{VE}(VE) = SQL_{VE}(E)$.

   c. If VE is a reference to the constant whose declaration is given by

      ```
      constant C [ : D ] is E ;
      ```

      then $SQL_{VE}(VE) = SQL_{VE}(E)$.

   d. If VE is a reference to a domain parameter P of domain D, and D assigns expression E to P (see rule 6 of section 4.1.3), then $SQL_{VE}(VE) = SQL_{VE}(E)$.

   e. If VE is a reference to the input parameter, INP, and $PARM_{SQL}(INP)$ is ": C T" (for C an *SQL*_identifier and T a data type, see section 5.6), then $SQL_{VE}(VE)$ is

      ```
      : C [ INDICATOR : INDIC_NAME(INP) ]
      ```

where INDICATOR $INDIC_{NAME}(INP)$ appears precisely when $INDIC_{SQL}(INP)$ is defined. See Section 5.6.

2. If VE is SF(VE1) where SF is a set function, then $SQL_{VE}(SF(VE_1))$ is $SF(SQL_{VE}(VE_1))$.

3. If VE is $D(VE_1)$, where D is a domain name, then $SQL_{VE}(D(VE_1))$ is $SQL_{VE}(VE_1)$.

4. If VE is $+VE_1$ (or $-VE_1$) then $SQL_{VE}(VE)$ is $+SQL_{VE}(VE_1)$ (or $-SQL_{VE}(VE_1)$).

5. If VE is VE_1 op VE_2 where op is an arithmetic operator, then $SQL_{VE}(VE)$ is $SQL_{VE}(VE_1)$ op $SQL_{VE}(VE_2)$.

6. If VE is (VE_1) then $SQL_{VE}(VE)$ is $(SQL_{VE}(VE_1))$.

Note: As a consequence of these definitions, particularly item 3, a domain conversion should be considered an instruction to a SAMeDL processor that a given expression is well-formed and should not be considered a data conversion. Although SAMeDL enforces a strict typing discipline, data conversions are carried out under the rules of SQL, not those of Ada. It is for this reason that warning messages are given for conversions which lose scale.

5.11 Search Conditions

search_condition ::= boolean_term | search_condition **or** boolean_term

boolean_term ::= boolean_factor | boolean_term **and** boolean_factor

boolean_factor ::= [**not**] boolean_primary

boolean_primary ::= predicate | (search_condition)

predicate ::= comparison_predicate |
between_predicate |
in_predicate |
like_predicate |
null_predicate |
quantified_predicate |
exists_predicate

The concrete syntax of search conditions differs from that of SQL only in that SAMeDL value expression (Section 5.10) replaces SQL value expression in the definition of the atomic predicates **[SQL]** 5.11 through 5.17. In addition, the SAMeDL enforces a strict typing discipline on the atomic predicates, not enforced by SQL.

For convenience, the following subsections present the syntax for each of the search predicates. Semantics are defined below in conjunction with **[SQL]**.

The atomic predicates of SQL take a varying number of operands; the comparison predicate takes two, the between predicate takes three, and the in predicate takes any number. So let $\{OP_1, OP_2, ..., OP_m\}$ be the set of operands of any atomic predicate. Each of the OP_i is of the form of a value expression. Therefore, the functions DOMAIN and DATACLASS may be applied to them (Section 5.10). For an atomic predicate to be well formed, then for any pair of distinct i and j, $1 \le i, j \le m$

1. If DOMAIN(OP_i) ≠ NO_DOMAIN and DOMAIN(OP_j) ≠ NO_DOMAIN, then DOMAIN(OP_i) = DOMAIN(OP_j), and

2. Exactly one of the following holds:

   a. DATACLASS(OP_i) = DATACLASS(OP_j) = **integer**;

   b. DATACLASS(OP_i) = DATACLASS(OP_j) = **character**;

   c. Both DATACLASS(OP_i) and DATACLASS(OP_j) are elements of the set {**fixed, float**}

   d. DATACLASS(OP_i) = DATACLASS(OP_j) = **enumeration**, and there exists some k, $1 \le k \le m$ such that

      i. DOMAIN(OP_k) ≠ NO_DOMAIN, and

      ii. For all l, $1 \le l \le m$, either

         1. DOMAIN(OP_l) = NO_DOMAIN, and OP_l is an enumeration literal of the domain DOMAIN(OP_k), or

         2. DOMAIN(OP_l) = DOMAIN(OP_k).

SQL Semantics

A SAMeDL search condition is transformed into an *SQL*_search_condition by application of the transformation SQL_{SC} which operates by executing the transformation SQL_{VE}, defined in Section 5.10, to the value expressions appearing within the search condition and the transformation SQL_{SQ}, defined in Section 5.12, to the subqueries in the search condition. In other words, let P, P_1, and P_2 be search conditions, VE, VE_1, VE_2, ..., VE_i be value expressions, and SQ be a subquery. Then $SQL_{SC}(P)$ is given by

1. If P is of the form: P_1 op P_2, where op is one of "**and**" or "**or**", then $SQL_{SC}(P)$ is $SQL_{SC}(P_1)$ op $SQL_{SC}(P_2)$ ([SQL] 5.18).

2. If P is of the form: "**not** P_1", then $SQL_{SC}(P)$ is **not** $SQL_{SC}(P_1)$ ([SQL] 5.18).

3. If P is of the form: "VE_1 op VE_2", where op is an SQL comparison operator ([SQL] 5.11), then $SQL_{SC}(P) = SQL_{VE}(VE_1)$ op $SQL_{VE}(VE_2)$. If P = VE op SQ, then $SQL_{SC}(P) = SQL_{VE}(VE)$ op $SQL_{SQ}(SQ)$ ([SQL] 5.11).

4. If P is of the form: "VE [**not**] **between** VE_1 **and** VE_2" then $SQL_{SC}(P) = SQL_{VE}(VE)$ [**not**] **between** $SQL_{VE}(VE_1)$ and $SQL_{VE}(VE_2)$ ([SQL] 5.12).

5. If P is of the form: "VE [**not**] **in** SQ", then $SQL_{SC}(P) = SQL_{VE}(VE)$ [**not**] **in** $SQL_{SQ}(SQ)$. If P is of the form: "VE [**not**] **in** (VE_1, VE_2, ..., VE_i, ...)" then $SQL_{SC}(P)$ = $SQL_{VE}(VE)$ [**not**] **in** ($SQL_{VE}(VE_1)$, $SQL_{VE}(VE_2)$, ..., $SQL_{VE}(VE_i)$,...)

6. If P is of the form: "VE_1 [**not**] **like** VE_2 **escape** c" where c is a character, then $SQL_{SC}(P) = SQL_{VE}(VE_1)$ [**not**] **like** $SQL_{VE}(VE_2)$ **escape** c ([SQL] 5.14).

7. If P is of the form: "C **is [not] null**" where C is a column reference, then $SQL_{SC}(P)$ = C **is [not] null** ([SQL] 5.15). ***Note:*** SQL_{SC} is the identity mapping on *SQL*_null_predicates.

8. If P is of the form: "VE op quant SQ" where op is an *SQL*_comp_op, quant is an *SQL*_qualifier (i.e., one of SOME, ANY or ALL), then $SQL_{SC}(P) = SQL_{VE}(VE)$ op quant $SQL_{SQ}(SQ)$ ([SQL] 5.16).

9. If P is of the form: "**exists** SQ", then $SQL_{SC}(P)$ = **exists** $SQL_{SQ}(SQ)$ ([SQL] 5.17).

5.11.1 Comparison Predicate

comparison_predicate ::= value_expression comp_op val_or_subquery

val_or_subquery ::= value_expression | subquery

comp_op ::= = | <> | < | > | <= | >=

5.11.2 Between Predicate

between_predicate ::= value_expression [**not**] **between** value_expression **and** value_expression

5.11.3 In Predicate

in_predicate ::= value_expression [**not**] **in** subquery_or_value_spec_list

subquery_or_value_spec_list ::= subquery | (value_spec_list)

value_spec_list ::= value_spec { , value_spec }

value_spec ::= input_reference |
static_expression |
user

5.11.4 Like Predicate

like_predicate ::= column_reference [**not**] **like** pattern_string [escape_clause]

pattern_string ::= value_spec

escape_clause ::= **escape** value_spec

5.11.5 Null Predicate

null_predicate ::= column_reference **is** [**not**] **null**

5.11.6 Quantified Predicate

quantified_predicate ::= value_expression comp_op quantifier subquery

quantifier ::= **all** | **some** | **any**

5.11.7 Exists Predicate

exists_predicate ::= **exists** subquery

5.12 Subqueries

The concrete syntax of a subquery ([SQL] 5.24) differs from that of query specifications in that the select list is limited to at most one parameter. Further, that parameter, when present, takes the form of a value expression (Section 5.10), not that of a select parameter (Section 5.7), as it is not visible to the user of the abstract module.

subquery ::= (**select** [**distinct** | **all**] result_expression
from_clause
[**where** search_condition]
[*SQL*_group_by_clause]
[**having** search_condition])

result_expression ::= value_expression | *

Ada Semantics

If, within a subquery, *SQ*, the result_expression takes the form of a value_expression, *VE*, then DOMAIN(SQ) = DOMAIN(VE) and DATACLASS(SQ) = DATACLASS(VE). DOMAIN(SQ) and DATACLASS(SQ) are undefined when the result_expression takes the form of *.

Note: The fact that DOMAIN(*) is undefined means that such a result_expression can be used only if the subquery appears within an exists_predicate.

SQL Semantics

The *SQL*_subquery formed from a SAMeDL subquery, *SQ*, denoted $SQL_{SQ}(SQ)$, is produced by removing any **as** keywords, if present, from the from_clause and applying the transformation SQL_{SC} to the search_conditions in the **where** and **having** clauses, if present.

5.13 Status Clauses

A status clause serves to attach a status map to a procedure and optionally rename the status parameter.

status_clause ::= **status** status_reference [named_phrase]

Ada Semantics

If a procedure P has a status_clause of the form

status M [**named** Id_1]

and the definition of M was given by (see Section 4.1.5):

enumeration T **is** (L_1, ..., L_n);

status M [**named** Id_2]
uses T

is (..., n=>L, ...);

(see section 4.1.8), then:

1. The procedure P_{Ada} (Sections 5.3 and 5.5) shall have a status parameter of type T.
2. The name of the status parameter of P_{Ada} is determined by:
    a. If id_1 is present in the status_clause, than the name of the status parameter shall be *id_1*.
    b. If rule(a) does not apply, then if id_2 is present in the definition of the status map M, the name of the status parameter shall be *id_2* (see section 4.1.8).
    c. If neither rule (a) nor rule (b) apply, then the name of the status parameter shall be *Status*.

Appendix A SAMeDL_Standard

The predefined SAMeDL definitional module SAMeDL_Standard provides a common location for declarations that are standard for all implementations of the SAMeDL. This definitional module includes the SAMeDL declarations for the status map Standard_Map and for the standard base domains.

```
definition module SAMeDL_Standard is

    exception SQL_Database_Error;
    exception Null_Value_Error;

    -- standard status map
    status Standard_Map named Is_Found uses boolean is
        (0 => True, 100 => False);

    -- SQL_Int is based on the Ada type SQL_Standard.Int
    base domain SQL_Int
        (first    : integer;
         last     : integer)
    is
        domain pattern is
            'type [self]_Not_Null is new SQL_Int_Not_Null'
                '{ range [first] .. [last] };'
            'type [self]_Type is new SQL_Int;'
            'package [self]_Ops is new SQL_Int_Ops ('
                '[self]_Type, [self]_Not_Null);'
        end pattern;

        derived domain pattern is
            'type [self]_Not_Null is new [parent]_Not_Null'
                '{ range [first] .. [last] };'
            'type [self]_Type is new [parent]_Type;'
            'package [self]_Ops is new SQL_Int_Ops ('
                '[self]_Type, [self]_Not_Null);'
        end pattern;

        subdomain pattern is
            'subtype [self]_Not_Null is [parent]_Not_Null'
                '{ range [first] .. [last] };'
            'type [self]_Type is new [parent]_Type;'
            'package [self]_Ops is new SQL_Int_Ops ('
                '[self]_Type, [self]_Not_Null);'
        end pattern;

        for not null type name use '[self]_Not_Null';
        for null type name use '[self]_Type';
        for data class use integer;
        for dbms type use integer;
        for conversion from dbms to not null use type mark;
        for conversion from not null to null use function
            '[self]_Ops.With_Null';
        for conversion from null to not null use function
            '[self]_Ops.Without_Null';
        for conversion from not null to dbms use type mark
```

```
    for context clause use 'with sql_int_pkg; use sql_int_pkg;' ;
    for null_bearing assign use '[self]_ops.assign' ;
    for not_null_bearing assign use predefined;
    for null_value use 'null_sql_int' ;

end SQL_Int;

-- SQL_Smallint is based on the Ada type SQL_Standard.Smallint
base domain SQL_Smallint
    (first     : integer;
     last      : integer)
is
    domain pattern is
        'type [self]_Not_Null is new SQL_Smallint_Not_Null'
            '( range [first] .. [last] );'
        'type [self]_Type is new SQL_Smallint;'
        'package [self]_Ops is new SQL_Smallint_Ops ('
            '[self]_Type, [self]_Not_Null);'
    end pattern;

    derived domain pattern is
        'type [self]_Not_Null is new [parent]_Not_Null'
            '( range [first] .. [last] );'
        'type [self]_Type is new [parent]_Type;'
        'package [self]_Ops is new SQL_Smallint_Ops ('
            '[self]_Type, [self]_Not_Null);'
    end pattern;

    subdomain pattern is
        'subtype [self]_Not_Null is [parent]_Not_Null'
            '( range [first] .. [last] );'
        'type [self]_Type is new [parent]_Type;'
        'package [self]_Ops is new SQL_Smallint_Ops ('
            '[self]_Type, [self]_Not_Null);'
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use integer;
    for dbms type use integer;
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
        '[self]_Ops.With_Null';
    for conversion from null to not null use function
        '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;

    for context clause use
        'with sql_smallint_pkg; use sql_smallint_pkg;' ;
    for null_bearing assign use '[self]_ops.assign' ;
    for not_null_bearing assign use predefined;
    for null_value use 'null_sql_smallint' ;

end SQL_Smallint;

-- SQL_Real is based on the Ada type SQL_Standard.Real
base domain SQL_Real
    (first     : float;
```

```
        last     : float)
is
    domain pattern is
        'type [self]_Not_Null is new SQL_Real_Not_Null'
            '( range [first] .. [last] );'
        'type [self]_Type is new SQL_Real;'
        'package [self]_Ops is new SQL_Real_Ops ('
            '[self]_Type, [self]_Not_Null);'
    end pattern;

    derived domain pattern is
        'type [self]_Not_Null is new [parent]_Not_Null'
            '( range [first] .. [last] );'
        'type [self]_Type is new [parent]_Type;'
        'package [self]_Ops is new SQL_Real_Ops ('
            '[self]_Type, [self]_Not_Null);'
    end pattern;

    subdomain pattern is
        'subtype [self]_Not_Null is [parent]_Not_Null'
            '( range [first] .. [last] );'
        'type [self]_Type is new [parent]_Type;'
        'package [self]_Ops is new SQL_Real_Ops ('
            '[self]_Type, [self]_Not_Null);'
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use float;
    for dbms type use real;
    for conversion from dbms to not null use type mark;
    for conversion from not null to null use function
        '[self]_Ops.With_Null';
    for conversion from null to not null use function
        '[self]_Ops.Without_Null';
    for conversion from not null to dbms use type mark;

    for context clause use
        'with sql_real_pkg; use sql_real_pkg;' ;
    for null_bearing assign use '[self]_ops.assign' ;
    for not_null_bearing assign use predefined;
    for null_value use 'null_sql_real' ;

end SQL_Real;

-- SQL_Double_Precision is based on the Ada type SQL_Standard.Double_Precision
base domain SQL_Double_Precision
    (first     : float;
     last      : float)
is
    domain pattern is
        'type [self]_Not_Null is new SQL_Double_Precision_Not_Null'
            '( range [first] .. [last] );'
        'type [self]_Type is new SQL_Double_Precision;'
        'package [self]_Ops is new SQL_Double_Precision_Ops ('
```

```
            '[self]_Type, [self]_Not_Null);'
        end pattern;

        derived domain pattern is
            'type [self]_Not_Null is new [parent]_Not_Null'
                '( range [first] .. [last] );'
            'type [self]_Type is new [parent]_Type;'
            'package [self]_Ops is new SQL_Double_Precision_Ops ('
                '[self]_Type, [self]_Not_Null);'
        end pattern;

        subdomain pattern is
            'subtype [self]_Not_Null is [parent]_Not_Null'
                '( range [first] .. [last] );'
            'type [self]_Type is new [parent]_Type;'
            'package [self]_Ops is new SQL_Double_Precision_Ops ('
                '[self]_Type, [self]_Not_Null);'
        end pattern;

        for not null type name use '[self]_Not_Null';
        for null type name use '[self]_Type';
        for data class use float;
        for dbms type use double precision;
        for conversion from dbms to not null use type mark;
        for conversion from not null to null use function
            '[self]_Ops.With_Null';
        for conversion from null to not null use function
            '[self]_Ops.Without_Null';
        for conversion from not null to dbms use type mark;

        for context clause use
            'with sql_double_precision_pkg; use sql_double_precision_pkg;' ;
        for null_bearing assign use '[self]_ops.assign' ;
        for not_null_bearing assign use predefined;
        for null_value use 'null_sql_double_precision' ;

    end SQL_Double_Precision;

    -- SQL_Char is based on the Ada type SQL_Standard.Char
    base domain SQL_Char
    is
        domain pattern is
            'type [self]NN_Base is new SQL_Char_Not_Null;'
            'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
            'type [self]_Base is new SQL_Char;'
            'subtype [self]_Type is [self]_Base ('
                '[self]_Not_Null"length);'
            'package [self]_Ops is new SQL_Char_Ops ('
                '[self]_Base, [self]NN_Base);'
        end pattern;

        derived domain pattern is
            'type [self]NN_Base is new [parent]NN_Base;'
```

```
            'subtype [self]_Not_Null is [self]NN_Base (1 .. [length]);'
            'type [self]_Base is new [parent]_Base;'
            'subtype [self]_Type is [self]_Base ('
                '[self]_Not_Null"length);'
            'package [self]_Ops is new SQL_Char_Ops ('
                '[self]_Base, [self]NN_Base);'
        end pattern;

        subdomain pattern is
            'subtype [self]NN_Base is [parent]NN_Base;'
            'subtype [self]_Not_Null is [parent]NN_Base (1 .. [length]);'
            'type [self]_Base is new [parent]_Base;'
            'subtype [self]_Type is [self]_Base ('
                '[self]_Not_Null"length);'
            'package [self]_Ops is new SQL_Char_Ops ('
                '[self]_Base, [self]NN_Base);'
        end pattern;

        for not null type name use '[self]_Not_Null';
        for null type name use '[self]_Type';
        for data class use character;
        for dbms type use character '([length])';
        for conversion from dbms to not null use type mark;
        for conversion from not null to null use function
            '[self]_Ops.With_Null';
        for conversion from null to not null use function
            '[self]_Ops.Without_Null';
        for conversion from not null to dbms use type mark;

        for context clause use 'with sql_char_pkg; use sql_char_pkg;' ;
        for null_bearing assign use predefined;
        for null value use 'null_sql_char' ;

    end SQL_Char;

    -- SQL_Enumeration_As_Int is based on the Ada type SQL_Standard.Int
    base domain SQL_Enumeration_As_Int
        (map := pos)
    is
        domain pattern is
            'type [self]_Not_Null is new [enumeration];'
            'package [self]_Pkg is new SQL_Enumeration_Pkg ('
                '[self]_not_null)'
            'type [self]_Type is new [self]_Pkg.SQL_Enumeration;'
        end pattern;

        derived domain pattern is
            'type [self]_Not_Null is new [parent]_Not_Null;'
            'type [self]_Type is new [parent]_Type;'
        end pattern;

        subdomain pattern is
            'subtype [self]_Not_Null is [parent]_Not_Null;'
            'subtype [self]_Type is [parent]_Type;'
        end pattern;

        for not null type name use '[self]_Not_Null';
```

```
        for null type name use '[self]_Type';
        for data class use enumeration;
        for dbms type use integer;
        for conversion from not null to null use function
            'With_Null';
        for conversion from null to not null use function
            'Without_Null';

        for context clause use 'with sql_enumeration_pkg;' ;
        for not_null_bearing assign use predefined;
        for null_bearing assign use 'Assign' ;
        for null value use 'null_sql_enumeration';

    end SQL_Enumeration_As_Int;

    -- SQL_Enumeration_As_Char is based on the Ada type SQL_Standard.Char
    base domain SQL_Enumeration_As_Char
        (width : integer;
         map := image)
    is
        domain pattern is
            'type [self]_Not_Null is new [enumeration];'
            'package [self]_Pkg is new SQL_Enumeration_Pkg ('
                '[self]_not_null)'
            'type [self]_Type is new [self]_Pkg.SQL_Enumeration;'
        end pattern;

        derived domain pattern is
            'type [self]_Not_Null is new [parent]_Not_Null;'
            'type [self]_Type is new [parent]_Type;'
        end pattern;

        subdomain pattern is
            'subtype [self]_Not_Null is [parent]_Not_Null;'
            'subtype [self]_Type is [parent]_Type;'
        end pattern;

        for not null type name use '[self]_Not_Null';
        for null type name use '[self]_Type';
        for data class use enumeration;
        for dbms type use character '([width])';
        for conversion from not null to null use function
            'With_Null';
        for conversion from null to not null use function
            'Without_Null';

        for context clause use 'with sql_enumeration_pkg;' ;
        for not_null_bearing assign use predefined;
        for null_bearing assign use 'Assign' ;
        for null value use 'null_sql_enumeration';

    end SQL_Enumeration_As_Char;

end SAMeDL_Standard;
```

Appendix B SAMeDL_System

The predefined SAMeDL definitional module SAMeDL_System provides a common location for the declaration of implementation-defined constants that are specific to a particular DBMS/Ada compiler platform.

```
definition module SAMeDL_System is

    -- Smallest (most negative) value of any integer type
    constant Min_Int is implementation defined;
    -- Largest (most positive) value of any integer type
    constant Max_Int is implementation defined;

    -- Smallest value of any SQL_Int type
    constant Min_SQL_Int is implementation defined;
    -- Largest value of any SQL_Int type
    constant Max_SQL_Int is implementation defined;

    -- Smallest value of any SQL_Smallint type
    constant Min_SQL_Smallint is implementation defined;
    -- Largest value of any SQL_Smallint type
    constant Max_SQL_Smallint is implementation defined;

    -- Largest value allowed for the number of significant decimal
    -- digits in any floating point constraint
    constant Max_Digits is implementation defined;

    -- Largest value allowed for the number of significant decimal
    -- digits in any SQL_Real floating point constraint
    constant Max_SQL_Real_Digits is implementation defined;

    -- Largest value allowed for the number of significant decimal
    -- digits in any SQL_Double_Precision floating point constraint
    constant Max_SQL_Double_Precision_Digits is implementation defined;

    -- Largest value allowed for the number of characters in a
    -- character string constraint
    constant Max_SQL_Char_Length is implementation defined;

    -- SQL Standard value for successful execution of an SQL DML statement
    constant Success is 0;
    -- SQL Standard value for data not found
    constant Not_Found is 100;

end SAMeDL_System;
```

Appendix C Standard Support Operations and Specifications

The following two sections discuss the SAME standard support packages. The first section describes how they support the standard base domains, and the second section lists their Ada package specifications.

C.1 Standard Base Domain Operations

The SAME standard support packages encapsulate the Ada type definitions of the standard base domains, as well as the operations that provide the data semantics for domains declared using these base domains. This section describes the nature of the support packages, namely the Ada data types and the operations on objects of these types.

The SQL standard package SQL_Standard contains the type definitions for a DBMS platform that define the Ada representations of the concrete SQL data types. A standard base domain exists in the SAMeDL for each type in SQL_Standard (except for SQLCode_Type), and these base domains are each supported by one of the SAME standard support packages, In addition to the above base domains, two standard base domains exist that provide data semantics for Ada enumeration types.

Each support package defines a not null-bearing and a null-bearing type for the base domain. The not null-bearing type is a visible Ada type derived from the corresponding type in SQL_Standard with no added constraints. This type provides the Ada application programmer with Ada data semantics for data in the database. The null-bearing type is an Ada limited private type used to support data semantics of the SQL null value. In particular, the null-bearing type may contain the null value; the not null-bearing type may not.

Domains are derived from base domains by the declaration of two Ada data types, derived from the types in the support packages, and the instantiation of the generic operations package with these types. The type derivations and the package instantiation provide the domain with the complete set of operations that define the data semantics for that domain. These operations are described below, grouped by data class.

C.1.1 All Domains

All domains derived from the standard base domains make an *Assign* procedure available to the application because the type that supports the SQL data semantics is an Ada limited private type. For the numeric domains, this procedure enforces the range constraints that are specified for the domain when it is declared. The Ada *Constraint_Error* exception is raised by these procedures if the value to be assigned falls outside of the specified range.

A parameterless function named *Null_SQL_<type>* is available for all domains as well. This function returns an object of the null-bearing type of the appropriate domain whose value is the SQL null value.

Every domain has a set of conversion functions available for converting between the not null-bearing type and the null-bearing type. The function *With_Null* converts an object of the not null-bearing type and the null-bearing type. The function *Without_Null* converts converts an object of the null-bearing type to an object of the no null-bearing type. *Without_Null* will raise

the *Null_Value_Error* exception if the value of the object that it is converting is the SQL null value, since an object of the not null-bearing type can never be null.

Two testing functions are available for each domain as well. The boolean functions *Is_Null* and *Not_Null* test objects of the null-bearing type, returning the appropriate boolean value indicating whether or not an object contains the SQL null value.

Additionally, all domains provide two sets of comparison operators that operate on objects of the null-bearing type. The first set of operators returns boolean values, and the second set of operators returns objects of the type Boolean_With_Unknown, defined in the support package SQL_Boolean_Pkg (see Section C.2.3), which implements three-valued logic. The boolean comparison operators are =, /=, <, >, <=, and >=, and return the value *False* if either of the objects contains the SQL null value. Otherwise, these operators perform the comparison, and return the appropriate boolean result. The *Boolean_With_Unknown* comparison operators are *Equals* and , <, >, <=, and >=, and return the value *Unknown* if either of the objects contains the SQL null value. Otherwise, these operators perform the comparison, and return the *Boolean_With_Unknown* values *True* or *False*.

C.1.2 Numeric Domains

In addition to the operations mentioned above, all numeric domains provide unary and binary arithmetics operations for the null-bearing type of the domain. The subprograms that implement these operations provide the data semantics of the SQL null value with respect to these arithmetic operations. Specifically, any arithmetic operation applied to a null value results in the null value. Otherwise, the operation is defined to be the same as the Ada operation. The unary operations that are provided are +, -, and *Abs*. The binary operations include +, -, *, and /. Finally, all numeric domains provide the exponentiation operation (**).

C.1.3 Int and Smallint Domains

Int and *Smallint* domains provide the application programmer with the Ada functions *Mod* and *Rem* that operate on objects of the null-bearing type. Again, the subprograms that implement these operations provide the data semantics of the SQL null value with respect to these arithmetic operations. As with the other arithmetic operation, *Mod* and *Rem* return the null value when applied to an object containing the null value. Otherwise, they are defined to be the same as the Ada operation.

These domains also make *Image* and *Value* functions available to the application programmer. Both of these functions are overloaded, meaning that there are *Image* and *Value* functions that operate on objects of both the not null-bearing and the null-bearing types of the domain. The *Image* function converts an object of an *Int* or *Smallint* domain to a character representation of the integer value. The *Value* function converts a character representation of an integer value to an object of an *Int* or *Smallint* domain. These functions perform the same operation as the Ada attribute functions of the same name, except that the character set of the character inputs and outputs is that of the underlying SQL_Standard.Char character set. If the *Image* and *Value* functions are applied to objects of the null-bearing type containing the null value, a null character object and a null integer object are returned respectively.

C.1.4 Character Domains

In addition to the operations provided by all domains, character domains provide the application programmer with some string manipulation and string conversion operations.

Character domains provide two string manipulation functions that operate on objects of the null-bearing type. The first one is the concatenation function (&). If either of the input character objects contains the null value, then the object returned contains the null value. Otherwise this operation is the same as the Ada concatenation operation. The other function is the *Substring* function, which is patterned after the substring function of SQL2. This function returns the portion of the input character object specified by the *Start* and *Length* index inputs. An Ada *Constraint_Error* is raised if the substring specification is not contained entirely within the input string.

The remaining operations provided by the character domains are conversion functions. A *To_String* and a *To_Unpadded_String* function exist for both the not null-bearing and the null-bearing types of the domain. The *To_String* function converts its input, which exists as an object whose value is comprised of characters from the underlying character set of the platform, to an object of the Ada predefined type *Standard.String*. If conversion of a null-bearing object containing the null value is attempted, the *Null_Value_Error* exception is raised. The *To_Unpadded_String* functions are identical in every way to the *To_String* functions except that trailing blanks are stripped from the value.

The *Without_Null_Unpadded* function is identical to the *Without_Null* function, described in section C.1.1 above, except that trailing blanks are stripped from the value.

Two functions exist that convert objects of the Ada predefined type *Standard.String* to objects of the not null-bearing and null-bearing types of the domain. The *To_SQL_Char_Not_Null* function converts an object of type *Standard.String* to the not null-bearing type of the domain. The *To_SQL_Char* function converts an object of type *Standard.String* to an object of the null-bearing type.

Finally, character domains provide the function *Unpadded_Length*, which returns the length of the character string representation without trailing blanks. This function operates on objects of the null-bearing type, and raises the *Null_Value_Error* exception if the input object contains the null value.

C.1.5 Enumeration Domains

Enumeration domains provide functions for the null-bearing type that are normally available as Ada attribute functions for the not null-bearing type. The *Image* and *Value* functions have the same semantics as described for *Int* and *Smallint* domains in Section C.1.3 above, except that they operate on enumeration values rather than integers.

The *Pred* and *Succ* functions operate on objects of the null-bearing type, and return the previous and next enumeration literals of the underlying enumeration type, respectively. If these functions are applied to objects containing the null value, an object containing the null value is returned.

The last two functions are the *Pos* and *Val* functions. These functions also operate on objects of the null-bearing type. *Pos* returns a value of the Ada predefined type *Standard.Integer* representing the position (relative to zero) of the enumeration literal that is the value of the input object. If the input object contains the null value, then the *Null_Value_Error* exception is raised. The *Val* function accepts a value of the predefined type *Standard.Integer* and returns the enumeration literal whose position in the underlying enumeration type is specified by that value. If the input integer value falls outside the range of available enumeration literals, the Ada *Constraint_Error* is raised.

C.1.6 Boolean Functions

The SAME standard support package SQL_Boolean_Pkg defines a number of boolean functions, namely *not, and, or,* and *xor,* which implement three-valued logic as defined in [SQL]. All of these functions operate on two input parameters of the type *Boolean_With_Unknown,* and return a value of that type.

This support package also provides a conversion function, which converts the input of the type *Boolean_With_Unknown* to a value of the Ada predefined type boolean. If the input object has the value *Unknown,* then the *Null_Value_Error* exception is raised.

Finally, the package provides three testing functions that return boolean values. These functions, *Is_True, Is_False,* and *Is_Unknown,* return the value true if the input passes the test; otherwise functions return the value false.

C.1.7 Operations Available to the Application

	Operand Type			Exceptions
	Left	**Right**	**Result**	
All Domains				
Null_SQL_<type>			_Type	
With_Null		_Not_Null	_Type	
Without_Null		_Type[1]	_Not_Null[2]	Null_Value_Error
Is_Null, Not_Null		_Type	Boolean	
Assign[3]	_Type	_Type		Constraint_Error
.....Equals, Not_Equals	_Type	_Type	B_W_U[4]	
<, >, <=, >=	_Type	_Type	B_W_U	
=, /=. >, <, >=, <=	_Type	_Type	Boolean	
Numeric Domains				
unary +/-, Abs		_Type	_Type	
+, -, /, *	_Type	_Type	_Type	
**	_Type	Integer	_Type	
Int and Smallint Domains				
Mod, Rem	_Type	_Type	_Type	
Image	_Type		SQL_Char	
Image	_Not_Null		SQL_ChrNN[5]	
Value	SQL_Char		_Type	
Value	SQL_ChrNN		_Not_Null	
Character Domains				
Without_Null_Unpadded		_Type	_Not_Null	Null_Value_Error
To_String		_Not_Null	String	
To_String		_Type	String	Null_Value_Error
To_Unpadded_String		_Not_Null	String	
To_Unpadded_String		_Type	String	Null_Value_Error
To_SQL_Char_Not_Null		String	_Not_Null	
To_SQL_Char		String	_Type	
Unpadded_Length		_Type	SQL_U_L[9]	Null_Value_Error
Substring[10]		_Type	_Type	Constraint_Error
&	_Type	_Type	_Type	

Enumeration Domains

Pred, Succ		_Type	_Type	
Image		_Type	SQL_Char	
Image		_Not_Null	SQL_ChrNN	
Pos		_Type	Integer	Null_Value_Error
Val		Integer	_Type	
Value		SQL_Char	_Type	
Value		SQL_ChrNN	_Not_Null	

Boolean Functions

not		B_W_U	Boolean	
and, or, xor	B_W_U	B_W_U	Boolean	
To_Boolean		B_W_U	Boolean	Null_Value_Error
Is_True	B_W_U	B_W_U	Boolean	
Is_False	B_W_U	B_W_U	Boolean	
Is_Unknown	B_W_U	B_W_U	Boolean	

1. "_Type" represents the type in the abstract domain of which objects that may be null are declared.

2. "_Not_Null" represents the type in the abstract domain of which objects that are not null may be declared.

3. "Assign" is a procedure. The result is returned in object "Left".

4. "B_W_U" is an abbreviation for Boolean_With_Unknown.

5. "SQL_ChrNN" is an abbreviation for SQL_Char_Not_Null.

9. "SQL_U_L" is an abbreviation for SQL_Char_Pkg subtype SQL_Unpadded_Length.

10. Substring has two additional parameters: Start and Length, which are both of the SQL_Char_Pkg subtype SQL_Char_Length.

C.2 Standard Support Package Specifications

C.2.1 SQL_Standard

The package SQL_Standard is defined in **[ESQL]** and is reproduced here for information only.

```
package Sql_Standard is

    package Character_Set renames csp;
    subtype Character_Type is Character_Set.cst;
    type Char is array (positive range <>)
        of Character_Type;
    type Smallint is range bs .. ts;
    type Int is range bi .. ti;
    type Real is digits dr;
    type Double_Precision is digits dd;
    type Sqlcode_Type is range bsc .. tsc;
    subtype Sql_Error is Sqlcode_Type range Sqlcode_Type'FIRST .. -1;
    subtype Not_Found is Sqlcode_Type range 100..100;
    subtype Indicator_Type is t;
```

```
-- csp is an implementator-defined package and cst is an
-- implementor-defined character type. bs, tw, bi, ti, dr, dd, bsc,
-- and tsc are implementor-defined integral values. t is int or
-- smallint corresponding to an implementor-defined <exact_numeric_type>
-- of indicator parameters.

end Sql_Standard;
```

C.2.3 SQL_Boolean_Pkg

```
package SQL_Boolean_Pkg is

    type Boolean_with_Unknown is (FALSE, UNKNOWN, TRUE);

    --| Three valued Logic operations
    --|  three-val X three-val => three-val
    --|
    --| A   B     A and B   A or B   A xor B   not A
    --| T   T     T         T        F         F
    --| T   F     F         T        T         F
    --| F   F     F         F        F         T
    --| T   U     U         T        U         F
    --| F   U     F         U        U         T
    --| U   U     U         U        U         U

    function "not"  (Left : Boolean_with_Unknown) return Boolean_with_Unknown;
    function "and" (Left, Right : Boolean_with_Unknown) return Boolean_with_Unknown;
    function "or" (Left, Right : Boolean_with_Unknown) return Boolean_with_Unknown;
    function "xor" (Left, Right : Boolean_with_Unknown) return Boolean_with_Unknown;

    --- three-val => bool or exception---
    function To_Boolean (Left : Boolean_with_Unknown) return Boolean;

    --- three-val => bool---
    function Is_True (Left : Boolean_with_Unknown) return Boolean;
    function Is_False (Left : Boolean_with_Unknown) return Boolean;
    function Is_Unknown (Left : Boolean_with_Unknown) return Boolean;

end SQL_Boolean_Pkg;
```

C.2.4 SQL_Int_Pkg

```
with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package SQL_Int_Pkg is

    type SQL_Int_not_null is new SQL_Standard.Int;

    --- Possibly Null Integer---
    type SQL_Int is limited private;

    function Null_SQL_Int return SQL_Int;
```

```
--| This pair of functions convert between the null-bearing and non-null-bearing types.

function Without_Null_Base(Value : SQL_Int) return SQL_Int_Not_Null;
function With_Null_Base(Value : SQL_Int_Not_Null) return SQL_Int;

    --| With_Null_Base raises Null_Value_Error if the input value is null


--| This procedure implements range checking. Note: it is not meant to be used directly
--| by application programmers. See the generic package SQL_Int_Ops.
--| Raises constraint_error if not (First <= Right <= Last)

procedure Assign_with_check (
                Left : in out SQL_Int; Right :  SQL_Int;
                First, Last : SQL_Int_Not_Null);


--| The following functions implement three valued arithmetic. If either input to any of
--| these functions is null, the function returns the null value; otherwise they perform
--| the indicated operation.  These functions raise no exceptions.

function "+"(Right : SQL_Int) return SQL_Int;
function "-"(Right : SQL_Int) return SQL_Int;
function "abs"(Right : SQL_Int) return SQL_Int;
function "+"(Left, Right : SQL_Int) return SQL_Int;
function "*"(Left, Right : SQL_Int) return SQL_Int;
function "-"(Left, Right : SQL_Int) return SQL_Int;
function "/"(Left, Right : SQL_Int) return SQL_Int;
function "mod" (Left, Right : SQL_Int) return SQL_Int;
function "rem" (Left, Right : SQL_Int) return SQL_Int;
function "**" (Left : SQL_Int; Right: Integer) return SQL_Int;


--| simulation of 'IMAGE and 'VALUE that return/take SQL_Char[_Not_Null] instead
--| of string

function IMAGE (Left : SQL_Int_Not_Null) return SQL_Char_Not_Null;
function IMAGE (Left : SQL_Int) return SQL_Char;
function VALUE (Left : SQL_Char_Not_NUll) return SQL_Int_Not_Null;
function VALUE (Left : SQL_Char) return SQL_Int;


--| Logical Operations
--|       type X type => Boolean_with_unknown
--|
--|       These functions implement three valued logic.  If either input is the null value,
--|       the functions return the truth value UNKNOWN; otherwise they perform the
--|       indicated comparison.  These functions raise no exceptions.

function Equals (Left, Right : SQL_Int) return Boolean_with_Unknown;
function Not_Equals (Left, Right : SQL_Int) return Boolean_with_Unknown;
function "<" (Left, Right : SQL_Int) return Boolean_with_Unknown;
function ">" (Left, Right : SQL_Int) return Boolean_with_Unknown;
function "<=" (Left, Right : SQL_Int) return Boolean_with_Unknown;
function ">=" (Left, Right : SQL_Int) return Boolean_with_Unknown;

--|       type => boolean
function Is_Null(Value : SQL_Int) return Boolean;
```

```
        function Not_Null(Value : SQL_Int) return Boolean;

        --| These functions of class type => boolean Equate UNKNOWN with FALSE. That is,
        --| they return TRUE only when the function returns TRUE. UNKNOWN and FALSE
        --| are mapped to FALSE.

        function "=" (Left, Right : SQL_Int) return Boolean;
        function "<" (Left, Right : SQL_Int) return Boolean;
        function ">" (Left, Right : SQL_Int) return Boolean;
        function "<=" (Left, Right : SQL_Int) return Boolean;
        function ">=" (Left, Right : SQL_Int) return Boolean;


        --| This generic is instantiated once for every abstract domain based on the SQL type
        --| Int. The three subprogram formal parameters are meant to default to the programs
        --| declared above. That is, the package should be instantiated in the scope of a use
        --| clause for SQL_Int_Pkg. The two actual types together form the abstract domain.
        --| The purpose of the generic is to create functions which convert between the two
        --| actual types and a procedure which implements a range constrained assignment for
        --| the null-bearing type. The bodies of these subprograms are calls to subprograms
        --| declared above and passed as defaults to the generic.

    generic
        type With_Null_type is limited private;
        type Without_null_type is range <>;
        with function With_Null_Base(Value : SQL_Int_Not_Null) return With_Null_Type is <>;
        with function Without_Null_Base(Value : With_Null_Type) return SQL_Int_Not_Null is <>;
        with procedure Assign_with_check (
                                Left : in out With_Null_Type;
                                Right : With_Null_Type;
                                First, Last : SQL_Int_Not_Null) is <>;
    package SQL_Int_Ops is
        function With_Null (Value : Without_Null_type) return With_Null_type;
        function Without_Null (Value : With_Null_Type) return Without_Null_type;
        procedure Assign (Left : in out With_null_Type; Right : in With_null_type);
    end SQL_Int_Ops;

    private

        --| not shown

    end SQL_Int_Pkg;
```

C.2.5 SQL_Smallint_Pkg

```
    with SQL_Standard;
    with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
    with SQL_Char_Pkg; use SQL_Char_Pkg;
    package SQL_Smallint_Pkg is

        type SQL_Smallint_not_null is new SQL_Standard.Smallint;

        ---- Possibly Null Integer----
        type SQL_Smallint is limited private;

        function Null_SQL_Smallint return SQL_Smallint;
```

```
--| this pair of functions converts between the null-bearing and non-null-bearing types

function Without_Null_Base(Value : SQL_Smallint) return SQL_Smallint_Not_Null;
function With_Null_Base(Value : SQL_Smallint_Not_Null) return SQL_Smallint;

    --| With_Null_Base raises Null_Value_Error if the input value is null


--| This procedure implements range checking. Note: it is not meant to be used directly
--| by application programmers. See the generic package SQL_Smallint_Ops.
--| Raises constraint_error if not (First <= Right <= Last)

procedure Assign_with_check (
                Left : in out SQL_Smallint;
                Right : SQL_Smallint;
                First, Last : SQL_Smallint_Not_Null);


--| The following functions implement three valued arithmetic. If either input to any of
--| these functions is null, the function returns the null value; otherwise they perform
--| the indicated operation. These functions raise no exceptions.

function "+"(Right : SQL_Smallint) return SQL_Smallint;
function "-"(Right : SQL_Smallint) return SQL_Smallint;
function "abs"(Right : SQL_Smallint) return SQL_Smallint;
function "+"(Left, Right : SQL_Smallint) return SQL_Smallint;
function "*"(Left, Right : SQL_Smallint) return SQL_Smallint;
function "-"(Left, Right : SQL_Smallint) return SQL_Smallint;
function "/"(Left, Right : SQL_Smallint) return SQL_Smallint;
function "mod" (Left, Right : SQL_Smallint) return SQL_Smallint;
function "rem" (Left, Right : SQL_Smallint) return SQL_Smallint;
function "**" (Left : SQL_Smallint; Right: Integer) return SQL_Smallint;


--| simulation of 'IMAGE and 'VALUE that return/take SQL_Char[_Not_Null] instead
--| of string

function IMAGE (Left : SQL_Smallint_Not_Null) return SQL_Char_Not_Null;
function IMAGE (Left : SQL_Smallint) return SQL_Char;
function VALUE (Left : SQL_Char_Not_Null) return SQL_Smallint_Not_Null;
function VALUE (Left : SQL_Char) return SQL_Smallint;

--| Logical Operations
--|       type X type => Boolean_with_unknown
--|
--|       These functions implement three valued logic. If either input is the null value,
--|       the functions return the truth value UNKNOWN; otherwise they perform the
--|       indicated comparison. These functions raise no exceptions.

function Equals (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
function Not_Equals (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
function "<" (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
function ">" (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
function "<=" (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
function ">=" (Left, Right : SQL_Smallint) return Boolean_with_Unknown;

--| type => boolean
```

```
        function Is_Null(Value : SQL_Smallint) return Boolean;
        function Not_Null(Value : SQL_Smallint) return Boolean;

        --| These functions of class type => boolean. Equate UNKNOWN with FALSE. That is,
        --| they return TRUE only when the function returns TRUE. UNKNOWN and FALSE

        function "=" (Left, Right : SQL_Smallint) return Boolean;
        function "<" (Left, Right : SQL_Smallint) return Boolean;
        function ">" (Left, Right : SQL_Smallint) return Boolean;
        function "<=" (Left, Right : SQL_Smallint) return Boolean;
        function ">=" (Left, Right : SQL_Smallint) return Boolean;

        --| This generic is instantiated once for every abstract domain based on the SQL type
        --| Smallint. The three subprogram formal parameters are meant to default to the
        --| programs declared above. That is, the package should be instantiated in the scope
        --| of a use clause for SQL_Smallint_Pkg. The two actual types together form the
        --| abstract domain. The purpose of the generic is to create functions which convert
        --| between the two actual types and a procedure which implements a range
        --| constrained assignment for the null-bearing type. The bodies of these subprograms
        --| are calls to subprograms declared above and passed as defaults to the generic.

    generic
        type With_Null_type is limited private;
        type Without_null_type is range <>;
        with function With_Null_Base(Value : SQL_Smallint_Not_Null) return With_Null_Type is <>;
        with function Without_Null_Base(Value : With_Null_Type)
                        return SQL_Smallint_Not_Null is <>;
        with procedure Assign_with_check (
                            Left : in out With_Null_Type;
                            Right : With_Null_Type;
                            First, Last : SQL_Smallint_Not_Null) is <>;

    package SQL_Smallint_OPs is
        function With_Null (Value : Without_Null_type) return With_Null_type;
        function Without_Null (Value : With_Null_Type) return Without_Null_type;
        procedure Assign (Left : in out With_null_Type; Right : in With_null_type);
    end SQL_Smallint_ops;

    private

        --| not shown

    end SQL_Smallint_Pkg;
```

C.2.6 SQL_Real_Pkg

```
    with SQL_Standard;
    with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
    package SQL_Real_Pkg is

        type SQL_Real_Not_Null is new SQL_Standard.Real;

        --- Possibly Null Real---
        type SQL_Real is limited private;

        function Null_SQL_Real return SQL_Real;
```

```
--| this pair of functions converts between the null-bearing and non-null-bearing types

function Without_Null_Base(Value : SQL_Real) return SQL_Real_Not_Null;
function With_Null_Base(Value : SQL_Real_Not_Null) return SQL_Real;

    --| With_Null_Base raises Null_Value_Error if the input value is null


--| This procedure implements range checking. Note: it is not meant to be used directly
--| by application programmers. See the generic package SQL_Real_Ops.
--| Raises constraint_error if not (First <= Right <= Last)

procedure Assign_with_Check (
                Left : in out SQL_Real;
                Right : SQL_Real;
                First, Last : SQL_Real_Not_Null);

--| The following functions implement three valued arithmetic. If either input to any of
--| these functions is null, the function returns the null value; otherwise they perform
--| the indicated operation. These functions raise no exceptions.

function "+"(Right : SQL_Real) return SQL_Real;
function "-"(Right : SQL_Real) return SQL_Real;
function "abs"(Right : SQL_Real) return SQL_Real;
function "+"(Left, Right : SQL_Real) return SQL_Real;
function "*"(Left, Right : SQL_Real) return SQL_Real;
function "-"(Left, Right : SQL_Real) return SQL_Real;
function "/"(Left, Right : SQL_Real) return SQL_Real;
function "**"(Left : SQL_Real; Right : Integer) return SQL_Real;

--| Logical Operations
--|       type X type => Boolean_with_unknown
--|
--|       These functions implement three valued logic. If either input is the null value,
--|       the functions return the truth value UNKNOWN; otherwise they perform the
--|       indicated comparison. These functions raise no exceptions.

function Equals (Left, Right : SQL_Real) return Boolean_with_Unknown;
function Not_Equals (Left, Right : SQL_Real) return Boolean_with_Unknown;
function "<" (Left, Right : SQL_Real) return Boolean_with_Unknown;
function ">" (Left, Right : SQL_Real) return Boolean_with_Unknown;
function "<=" (Left, Right : SQL_Real) return Boolean_with_Unknown;
function ">=" (Left, Right : SQL_Real) return Boolean_with_Unknown;

--| type => boolean
function Is_Null(Value : SQL_Real) return Boolean;
function Not_Null(Value : SQL_Real) return Boolean;


--| These functions of class type => boolean
--| Equate UNKNOWN with FALSE. That is, they return TRUE only when the function
--| returns TRUE. UNKNOWN and FALSE are mapped to FALSE.

function "=" (Left, Right : SQL_Real) return Boolean;
function "<" (Left, Right : SQL_Real) return Boolean;
function ">" (Left, Right : SQL_Real) return Boolean;
function "<=" (Left, Right : SQL_Real) return Boolean;
function ">=" (Left, Right : SQL_Real) return Boolean;
```

```
        --| This generic is instantiated once for every abstract domain based on the SQL type
        --| Real. The three subprogram formal parameters are meant to default to the programs
        --| declared above. That is, the package should be instantiated in the scope of a use
        --| clause for SQL_Real_Pkg. The two actual types together form the abstract domain.
        --| The purpose of the generic is to create functions which convert between the two
        --| actual types and a procedure which implements a range constrained assignment for
        --| the null-bearing type. The bodies of these subprograms are calls to subprograms
        --| declared above and passed as defaults to the generic.

    generic
        type With_Null_type is limited private;
        type Without_null_type is digits <>;
        with function With_Null_Base(Value : SQL_Real_Not_Null)  return With_Null_Type is <>;
        with function Without_Null_Base(Value : With_Null_Type) return SQL_Real_Not_Null is <>;
        with procedure Assign_with_check (
                                Left : in out With_Null_Type;
                                Right : With_Null_Type;
                                First, Last : SQL_Real_Not_Null) is <>;

    package SQL_Real_Ops is
        function With_Null (Value : Without_Null_type) return With_Null_type;
        function Without_Null (Value : With_Null_Type) return Without_Null_type;
        procedure Assign (Left : in out With_Null_Type; Right : in With_Null_type);
    end SQL_Real_Ops;

    private

        --| not shown

    end SQL_Real_Pkg;
```

C.2.7 SQL_Double_Precision_Pkg

C.2.8 SQL_Char_Pkg

```
    with SQL_System; use SQL_System;
    with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
    with SQL_Standard;

    package SQL_Char_Pkg is

        subtype SQL_Char_Length is natural  range 1 .. MAXCHRLEN;
        subtype SQL_Unpadded_Length is natural range 0 .. MAXCHRLEN;

        type SQL_Char_Not_Null is new SQL_Standard.Char;
        type SQL_Char(Length : SQL_Char_Length) is limited private;

        function Null_SQL_Char return SQL_Char;

        --| The next three functions convert between null-bearing and non null-bearing-types.
        --| Without_Null_Base and With_Null_Base are inverses (mod. null values).
        --| See also SQL_Char_Ops generic package below

        function With_Null_Base(Value : SQL_Char_Not_Null) return SQL_Char;
        function Without_Null_Base(Value : SQL_Char) return SQL_Char_Not_Null;
```

```
function Without_Null_Unpadded_Base(Value : SQL_Char) return SQL_Char_Not_Null;

    --| Without_Null_Unpadded_Base removes trailing blanks from the input
    --| Axiom: unpadded_Length(x) = Without_Null_Unpadded_Base(x)'Length
    --| Both Without_Null_Base and Without_Null_Unpadded_Base raise
    --|  null_value_error if x is null

function Unpadded_Length (Value : SQL_Char) return SQL_Unpadded_Length;

--| The next six functions convert between Standard.String
--|  types and the SQL_Char and SQL_Char_Not_Null types

function To_String (Value : SQL_Char_Not_Null) return String;
function To_String (Value : SQL_Char) return String;
function To_Unpadded_String (Value : SQL_Char_Not_Null) return String;
function To_Unpadded_String (Value : SQL_Char) return String;
function To_SQL_Char_Not_Null (Value : String) return SQL_Char_Not_Null;
function To_SQL_Char (Value : String) return SQL_Char;

--| Assignment operator for "null-bearing" type

procedure Assign (Left : out SQL_Char; Right : SQL_Char);

--| Substring(x,k,m) returns the substring of x starting at position k (relative to 1) with
--| length m. Returns null value if x is null
--| Raises constraint_error if Start < 1 or Length < 1 or Start + Length - 1 > x.Length

function Substring (Value : SQL_Char; Start, Length : SQL_Char_Length) return SQL_Char;

--| "&" returns null if either parameter is null; otherwise performs concatenation in the
--| usual way, preserving all blanks. May raise constraint_error implicitly if result is too
--| large (i.e., greater than SQL_Char_Length'Last

function "&" (Left, Right : SQL_Char) return SQL_Char;

--| Logical Operations
--|       type X type => Boolean_with_unknown--
--|       The comparison operators return the boolean value UNKNOWN if either
--|       parameter is null; otherwise, the comparison is done in accordance with
--|       ANSI X3.135-1986 para 5.11 general rule 5; that is, the shorter of the two string
--|       parameters is effectively padded with blanks to be the length of the longer
--|       string and a standard Ada comparison is then made

function Equals (Left, Right : SQL_Char) return Boolean_with_Unknown;
function Not_Equals (Left, Right : SQL_Char) return Boolean_with_Unknown;
function "<" (Left, Right : SQL_Char) return Boolean_with_Unknown;
function ">" (Left, Right : SQL_Char) return Boolean_with_Unknown;
function "<=" (Left, Right : SQL_Char) return Boolean_with_Unknown;
function ">=" (Left, Right : SQL_Char) return Boolean_with_Unknown;

--| type => boolean--

function Is_Null(Value : SQL_Char) return Boolean;
function Not_Null(Value : SQL_Char) return Boolean;

--| These functions of class type => boolean equate UNKNOWN with FALSE. That is,
--| they return TRUE only when the function returns TRUE. UNKNOWN and FALSE
--| are mapped to FALSE.
```

```
        function "=" (Left, Right : SQL_Char) return Boolean;
        function "<" (Left, Right : SQL_Char) return Boolean;
        function ">" (Left, Right : SQL_Char) return Boolean;
        function "<=" (Left, Right : SQL_Char) return Boolean;
        function ">=" (Left, Right : SQL_Char) return Boolean;

        --| The purpose of the following generic is to generate conversion functions between a
        --| type derived from SQL_Char_Not_Null, which are effectively Ada strings and a type
        --| derived from SQL_Char, which mimic the behavior of SQL strings. The subprogram
        --| formals are meant to default; that is, this generic should be instantiated in the scope
        --| of an use clause for SQL_Char_Pkg.

    generic
        type With_Null_Type is limited private;
        type Without_Null_Type is array (positive range <>) of sql_standard.Character_type;
        with function With_Null_Base (Value: SQL_Char_Not_Null) return With_Null_Type is <>;
        with function Without_Null_Base (Value: With_Null_Type) return SQL_Char_Not_Null is <>;
        with function Without_Null_Unpadded_Base (Value: With_Null_Type)
                        return SQL_Char_Not_Null is <>;

    package SQL_Char_Ops is
        function With_Null (Value : Without_Null_Type) return With_Null_Type;
        function Without_Null (Value : With_Null_Type) return Without_Null_Type;
        function Without_Null_Unpadded (Value : With_Null_Type) return Without_Null_Type;
    end SQL_Char_Ops;

    private

        --| not shown

    end SQL_Char_Pkg;
```

C.2.9 SQL_Enumeration_Pkg

```
    with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
    with SQL_Char_Pkg; use SQL_Char_Pkg;
    generic
        type SQL_Enumeration_Not_Null is (<>);

    package SQL_Enumeration_Pkg is

        --| ---- Possibly Null Enumeration----
        type SQL_Enumeration is limited private;

        function Null_SQL_Enumeration return SQL_Enumeration;

        --| This pair of functions convert between the null-bearing and non-null-bearing types.

        function Without_Null(Value : in SQL_Enumeration) return SQL_Enumeration_Not_Null;
        function With_Null(Value : in SQL_Enumeration_Not_Null) return SQL_Enumeration;

            --| With_Null raises Null_Value_Error if the input value is null


        --| Assignment operator for "null-bearing" type
```

```
    procedure Assign ( Left : in out SQL_Enumeration; Right : in SQL_Enumeration);

    --| Logical Operations
    --|      type X type => Boolean_with_unknown
    --|      These functions implement three valued logic. If either input is the null value,
    --|      the functions return the truth value UNKNOWN; otherwise they perform the
    --|      indicated comparison. These functions raise no exceptions

    function Equals (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;
    function Not_Equals (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;
    function "<" (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;
    function ">" (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;
    function "<=" (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;
    function ">=" (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;

    --| type => boolean
    function Is_Null (Value : SQL_Enumeration) return Boolean;
    function Not_Null (Value : SQL_Enumeration) return Boolean;
    function "=" (Left, Right : SQL_Enumeration) return Boolean;
    function "<" (Left, Right : SQL_Enumeration) return Boolean;
    function ">" (Left, Right : SQL_Enumeration) return Boolean;
    function "<=" (Left, Right : SQL_Enumeration) return Boolean;
    function ">=" (Left, Right : SQL_Enumeration) return Boolean;

    --| 'Pred, 'Succ, 'Image, 'Pos, 'Val, and 'Value attributes of the
    --| SQL_Enumeration_Not_Null type, passed in, for the associated SQL_Enumeration
    --| (null) type. They all raise the Null_Value_Error exception if a null value is passed in.
    --|
    --| Pred raises the Constraint_Error exception if the value passed in is equal to
    --|      SQL_Enumeration_Not_Null'Last.
    --| Succ raises the Constraint_Error exception if the value passed in is equal to
    --|      SQL_Enumeration_Not_Null'First.
    --| Val raises the Constraint_Error exception if the value passed in is not in the range
    --|      P'POS(P'FIRST)..P'POS(P'LAST) for type P.
    --| Value raises the Constraint_Error exception if the sequence of characters passed in
    --|      does not have the syntax of an enumeration literal for the instantiated
    --|      enumeration type.

    function Pred (Value : in SQL_Enumeration) return SQL_Enumeration;
    function Succ (Value : in SQL_Enumeration) return SQL_Enumeration;
    function Pos (Value : in SQL_Enumeration) return Integer;
    function Image (Value : in SQL_Enumeration) return SQL_Char;
    function Image (Value : in SQL_Enumeration_Not_Null) return SQL_Char_Not_Null;
    function Val (Value : in Integer) return SQL_Enumeration;
    function Value (Value : in SQL_Char) return SQL_Enumeration;
    function Value (Value : in SQL_Char_Not_Null) return SQL_Enumeration_Not_Null;

private

    --| not shown

end SQL_Enumeration_Pkg;
```

Appendix D Transform Chart

Function	Section	Input	Output	Output Is
AdaNAME	4.1.5	Record Declaration	Ada Identifier	Default name of the row record formal parameter
	5.6	Parameter	Ada Identifier	Name of the parameter in the Ada procedure declaration
	5.7	Select Parameter	Ada Identifier	Name of the component in the Ada row record type
	5.8	Insert Column Specification	Ada Identifier	Name of the component in the Ada row record type
	5.10	Value Expression	Ada Identifier	Default name for record component if the expression appears as a select parameter
AdaType	4.1.4	Constant Declaration	Ada Identifier	Name of the type in Ada declaration of constant
	5.6	Parameter	Ada Identifier	Name of the type of the parameter in the Ada procedure declaration
	5.7	Select Parameter	Ada Identifier	Name of the type of the component in the Ada row record type
	5.8	Insert Column Specification	Ada Identifier	Name of the type of the component in the Ada row record type
$COMP_{Ada}$	5.2	Select Parameter Insert Column Specification	Ada Record Component	Used in declaration of the Ada row record type
	5.4	Select Parameter	Ada Record Component	Used in declaration of the Ada row record type
DATACLASS	2.4	Literal	Data Class	Data class of the literal
	4.1.1	Base Domain	Data Class	Data class of the base domain
	4.1.3	Domain Domain Parameter	Data Class	Data class of the domain or domain parameter
	4.1.4	Constant	Data Class	Data class of the constant
	4.2	Column	Data Class	Data class of the column
	5.6	Parameter	Data Class	Data class of the parameter
	5.10	Value Expression	Data Class	Data class of the expression
DBLngNAME	5.7 4.1.5	Dblength Phrase	Ada Identifier	Name of dblength parameter undefined if no dblength phrase
$DBLeng_{Ada}$	5.2 5.4	Select Parameter	Ada Record Component	Row record component used for dblength data
DBMS_TYPE	4.1.3	Domain	SQL Data Type	SQL data type to be used at the database interface with an object of the specified domain

Function	Section	Input	Output	Output Is
DOMAIN	4.1.3	Domain Parameter	NO_DOMAIN	NO_DOMAIN is the domain of a domain parameter
	4.1.4	Constant	Domain	Domain of the constant (NO_DOMAIN for universal constants)
	4.2.1	Column	Domain	Domain of the column
	5.6	Parameter	Domain	Domain of the parameter
	5.10	Value Expression	Domain	Domain of the expression (NO_DOMAIN for universal constants)
INDIC$_{NAME}$	5.6	Parameter	SQL Identifier	Name of an SQL indicator parameter
	5.7	Select Parameter	SQL Identifier	Name of an SQL indicator parameter
	5.8	Insert Column Specification	SQL Identifier	Name of an SQL indicator parameter
INDIC$_{SQL}$	5.6	Parameter	SQL Indicator Param	Name of an SQL indicator parameter
	5.7	Select Parameter	SQL Indicator Param	Name of an SQL indicator parameter
	5.8	Insert Column Specification	SQL Indicator Param	Name of an SQL indicator parameter
LENGTH	2.4	Literal	Natural Number	Length of the literal (NO_LENGTH if not char literal)
	4.1.3	Domain Domain Parameter	Natural Number	Length of objects in domain or domain parameter value (NO_LENGTH if not char domain or parameter)
	4.1.4	Constant	Natural Number	Length of the constant (NO_LENGTH if not char literal)
	4.2.1	Column	Natural Number	Length of domain of column
	5.6	Parameter	Natural Number	Length of domain of parameter
	5.10	Value Expression	Natural Number	Length of expression (NO_LENGTH if not char literal)
MODE	5.6	Parameter	Ada Mode	Mode of the parameter in the Ada procedure declaration
PARM$_{Ada}$	5.6	Parameter	Ada Parameter Declaration	Ada parameter declaration in the Ada procedure declaration
PARM$_{Row}$	5.9	Into Clause Insert From Clause	Ada Identifier	Name of row record parameter
PARM$_{SQL}$	5.6	Parameter	SQL Parameter	An SQL parameter declaration
	5.7	Select Parameter	SQL Parameter	An SQL parameter declaration
	5.8	Insert Column Specification	SQL Parameter	An SQL parameter declaration

Function	Section	Input	Output	Output Is
SCALE	2.4	Literal	Natural Number	Scale of the literal (NO_SCALE if not a numeric literal)
	4.1.3	Domain Domain Parameter	Natural Number	Scale of objects in domain or domain parameter value (NO_SCALE if not a numeric domain or parameter)
	4.1.4	Constant	Natural Number	Scale of the constant (NO_SCALE if not a numeric literal)
	4.2.1	Column	Natural Number	Scale of domain of column
	5.6	Parameter	Natural Number	Scale of domain of parameter
	5.10	Value Expression	Natural Number	Scale of expression using SQL rules (NO_SCALE if not a numeric literal)
SQL_{NAME}	2.3	Ada Identifier	SQL Identifier	Unique SQL identifier
	5.4	Cursor Name	SQL Identifier	Unique SQL cursor name
	5.6	Parameter Name	SQL Identifier	Unique SQL parameter name
	5.7	Select Parameter	SQL Identifier	Unique name of SQL parameter
	5.8	Column Name	SQL Identifier	Unique name of SQL parameter
SQL_{SC}	5.11	Search Condition	SQL Search Condition	An SQL search condition
SQL_{SQ}	5.12	Subquery	SQL Subquery	An SQL subquery
SQL_{VE}	5.10	Value Expression	SQL Value Expression	An SQL value expression
$TYPE_{Row}$	5.9	Into Clause Insert From Clause	Ada Identifier	Name of the type of the row record parameter
VALUE	4.1.4	Static Expression	Ada Literal	Value assigned to the expression by the rules of SQL

Appendix E Glossary

Abstract Interface. A set of Ada package specifications containing the type and procedure declarations to be used by an Ada application program to access the database.

Abstract Module. A module that specifies the database routines needed by an Ada application.

Assignment context. A value expression appears in an assignment context if the value of that value expression is to be implicitly or explicitly assigned to an object. The assignment contexts are: select parameters, constant declarations, values in a VALUES list of an insert statement, set items in an update statement.

Base domain. A template for defining domains.

Conform. A value expression in an assignment context conforms to a target domain if the rules of SQL allow the assignment of a value of the data class of the expression to an object of the data class of the domain.

Conversion method. A method of converting non-null data between objects of the not null-bearing type, the null-bearing type, and the database type associated with the domain.

Correlation name. See [SQL], Section 5.7.

Cursor. See [SQL], Section 8.3.

Data class. The data class of a value is either character, integer, fixed, float, or enumeration. The data class of a domain determines which values may be converted, implicitly or explicitly, to the domain.

Database type. The SQL data type to be used with an object of that domain when it appears in an SQL parameter declaration. This need not be the same as the type of the data as is stored in the database.

Definitional module. A module that contains shared definitions: that is, declarations of base domains, domains, subdomains, constants, records, exceptions, enumerations, and status maps that are used by other modules.

Domain. The set of values and applicable operations for objects associated with a domain. A domain is similar to Ada type.

Exposed. The exposed name of a module (table) that appears in a context clause (table ref) containing an as phrase (correlation name) is the identifier in the associated as phrase (correlation name); the module name (table name) is hidden. If the as phrase (correlation name) is not present, the exposed name is that module's (table's) name. The exposed name of a table or module is the name by which that table or module is referenced.

Extended. A table, view, module, procedure, cursor, or cursor procedure that includes some nonstandard operation or feature.

Hidden. See exposed.

Module. A definitional module, a schema module, or an abstract module.

Not null type. The Ada type associated with objects of a domain that may not take a null value.

Null type. The Ada type associated with objects of a domain that may take a null value.

Null value. SQL's means of recording missing information. A null value in a column indicates that nothing is known about the value that should occupy the column.

Options. The aspects of the base domain that are essential to the declaration of domains based upon the base domain. In particular they define the base domain's null- and not null-bearing type name, data class, database type, and conversion methods.

Patterns. A template used to create the Ada constructs that implement the Ada semantics of a domain, subdomain, or derived domain declaration.

Row record. The Ada record associated with procedures that contain either a fetch, select, or insert statement. It is used to transmit the database data to or from the client program.

Row record type. The Ada type of the row record.

SAME. SQL Ada Module Extensions.

SAMeDL. SQL Ada Module Description Language.

Schema module. The SAMeDL notion that corresponds to the SQL SCHEMA.

SQL. Structured Query Language.

SQLCODE. See [SQL], Section 7.3.

Standard Map. The Standard Map is a status map defined in SAMeDL_Standard that has the form "**status** Standard_Map **named** Is_Found **uses** boolean **is** (0 => true, 100 => false);". Standard_Map is the status map for fetch statements that appear in cursor declarations by default.

Standard post processing. The processing that occurs after the execution of an SQL procedure but before control is returned to the calling application.

Static expression. A value expression that can be evaluated at compile time (i.e., all the associated leaves consist solely of literals, constants, or domain parameter references).

Status map. A partial function that associates an enumeration literal or a raise statement with each specified list or range of SQLCODE values. Status maps are used within the abstract module to uniformly process the status data for all procedures.

Target domain. The domain of the object to which an assignment is being made in an *assignment context.*

Universal constant. A constant whose declaration does not contain a domain reference.

Value expression. A value expression differs from an SQL value expression in that (1) an operand may be a reference to a constant or a domain parameter, and (2) SAMeDL value expressions are strongly typed.

Appendix F Syntax Summary

This appendix provides a summary of the syntax for SAMeDL. Productions are ordered alphabetically by left-hand nonterminal name. The section number indicates the section where the production is given.

```
[ 5.1 ]
abstract_module ::=     [ context ]
                        [ extended ]
                        abstract module Ada_identifier_1 is
                            authorization schema_reference
                            { definition }
                            { procedure_or_cursor }
                        end [ Ada_identifier_2 ] ;

[ 5.10 ]
all_set_function ::= [ avg | max | min | sum ] ( [ all ] value_expression )

[ 3.2 ]
as_phrase ::= as Ada_identifier

[ 4.1.3 ]
bas_dom_ref ::= dom_ref | base_domain_reference

[ 4.1.1 ]
base_domain_declaration ::=     [ extended ] base domain Ada_identifier_1
                                    [ ( base_domain_parameter_list ) ]
                                is
                                    patterns
                                    options
                                end [ Ada_identifier_2 ] ;

[ 4.1.1.1 ]
base_domain_parameter ::=       Ada_identifier : data_class [ := static_expression ]    |
                                map := pos                                              |
                                map := image

[ 4.1.1 ]
base_domain_parameter_list ::= base_domain_parameter { ; base_domain_parameter }

[ 3.4 ]
base_domain_reference ::= [ module_reference . ] Ada_identifier

[ 5.11.2 ]
between_predicate ::= value_expression [ not ] between value_expression and value_expression

[ 5.11 ]
boolean_factor ::= [ not ] boolean_primary

[ 5.11 ]
boolean_primary ::= predicate | ( search_condition )

[ 5.11 ]
boolean_term ::= boolean_factor | boolean_term and boolean_factor
```

[4.2.1]
check_constraint_definition ::= **check** (search_condition)

[5.5]
close_statement ::= **close** [*Ada*_identifier]

[4.2.1]
column_constraint ::= **not null** *SQL*_unique_specification |
*SQL*_reference_specification |
check (search_condition)

[4.2.1]
column_definition ::= *SQL*_column_name [*SQL*_data_type]
[*SQL*_default_clause]
[column_constraint] : domain_reference

[3.4]
column_name ::= *SQL*_identifier

[3.4]
column_reference ::= [table_reference .] column_name

[5.3]
commit_statement ::= **commit work**

[5.11.1]
comp_op ::= = | <> | < | > | <= | >=

[5.11.1]
comparison_predicate ::= value_expression comp_op val_or_subquery

[3.1]
compilation_unit ::= module { module }

[4.1.5]
component ::= component_name [**dblength** [named_phrase]]

[4.1.5]
component_declaration ::= component { , component } : domain_reference [**not null**] ;

[4.1.5]
component_declarations ::= component_declaration { component_declaration }

[4.1.5]
component_name ::= *Ada*_identifier

[4.1.4]
constant_declaration ::= **constant** *Ada*_identifier [: domain_reference]
is static_expression ;

[3.4]
constant_reference ::= [module_reference .] *Ada*_identifier

[3.2]
context ::= context_clause { context_clause }

```
[ 3.2 ]
context_clause ::= with_clause | with_schema_clause | use_clause

[ 4.1.1.3 ]
converter ::=    function pattern_list    |
                 procedure pattern_list   |
                 type mark

[ 3.3 ]
correlation_name ::= SQL_identifier

[ 5.4 ]
cursor_declaration ::=   [ extended ] cursor Ada_identifier_1
                             [ input_parameter_list ]
                             for
                                 query
                                 [ SQL_order_by_clause ]
                             ;
                             [ is cursor_procedures
                         end [ Ada_identifier_2 ] ; ]

[ 5.5 ]
cursor_delete_statement ::=      delete from table_name
                                 [ where current of Ada_identifier ]

[ 3.4 ]
cursor_proc_reference ::= [ cursor_reference . ] Ada_identifier

[ 5.5 ]
cursor_procedure ::=     [ extended ] procedure Ada_identifier_1
                             [ input_parameter_list ]
                         is
                             cursor_statement
                             [ status_clause ]
                         ;

[ 5.5 ]
cursor_procedures ::= cursor_procedure { cursor_procedure }

[ 3.4 ]
cursor_reference ::= [ module_reference . ] Ada_identifier

[ 5.5 ]
cursor_statement ::=     open_statement              |
                         fetch_statement             |
                         close_statement             |
                         cursor_update_statement     |
                         cursor_delete_statement     |
                         extended_cursor_statement

[ 5.5 ]
cursor_update_statement ::=      update table_name
                                 set set_item { , set_item ]
                                 [ where current of Ada_identifier ]
```

[4.1.1.1]
data_class ::= **integer** |
character |
fixed |
float |
enumeration

[4.1.3]
database_mapping ::= enumeration_association_list | **pos** | **image**

[4.1.1.3]
dbms_type ::= **int** |
integer |
smallint |
real |
double precision |
char |
character |
implementation defined

[4.1]
definition ::= base_domain_declaration |
domain_declaration |
subdomain_declaration |
constant_declaration |
record_declaration |
enumeration_declaration |
exception_declaration |
status_map_declaration

[4.1]
definitional_module ::= [context]
[**extended**]
definition module *Ada*_identifier_1 **is**
{ definition }
end [*Ada*_identifier_2] ;

[5.3]
delete_statement ::= **delete from** table_name
[**where** search_condition]

[4.1.1.2]
derived_domain_pattern ::= **derived domain pattern is** pattern_list
end pattern ;

[5.10]
distinct_set_function ::= [**avg** | **max** | **min** | **sum** | **count**] (**distinct** column_reference)

[4.1.3]
dom_ref ::= domain_reference | subdomain_reference

[5.10]
domain_conversion ::= domain_reference (value_expression)

[4.1.3]
domain_declaration ::= **domain** *Ada*_identifier **is new** bas_dom_ref [**not null**]
[(parameter_association_list)] ;

[3.4]
domain_parameter_reference ::= domain_reference.*Ada*_identifier

[4.1.1.2]
domain_pattern ::= **domain pattern is** pattern_list
end pattern ;

[3.4]
domain_reference ::= [module_reference .] *Ada*_identifier

[4.1.3]
enumeration association ::= enumeration_literal => database_literal

[4.1.3]
enumeration_association_list ::= (enumeration_association { , enumeration_association })

[4.1.6]
enumeration_declaration ::= **enumeration** *Ada*_identifier_1 **is** (enumeration_literal_list) ;

[4.1.6]
enumeration_literal_list::= enumeration_literal { , enumeration_literal }

[3.4]
enumeration_literal_reference ::= [module_reference .] *Ada*_identifier

[3.4]
enumeration_reference ::= [module_reference .] *Ada*_identifier

[5.11.4]
escape_clause ::= **escape** value_spec

[4.1.7]
exception_declaration ::= **exception** *Ada*_identifier ;

[3.4]
exception_reference ::= [module_reference .] *Ada*_identifier

[5.11.7]
exists_predicate ::= **exists** subquery

[3.7]
extended_cursor_statement ::= *implementation defined*

[3.7]
extended_query_expression ::= *implementation defined*

[3.7]
extended_query_specification ::= *implementation defined*

[3.7]
extended_schema_element ::= *implementation defined*

[3.7]
extended_statement ::= *implementation defined*

[3.7]
extended_table_element ::= *implementation defined*

[5.10]
factor ::= [+ | -] primary

[5.5]
fetch_statement ::= **fetch** [*Ada*_identifier t] [into_clause]

[3.3]
from_clause ::= **from** table_ref { , table_ref }

[4.1.1.3]
fundamental ::= **for not null type name use** pattern_list ; |
for null type name use pattern_list ; |
for data class use data_class ; |
for dbms type use dbms_type [pattern_list] ; |
for conversion from type **to** type **use** converter ;

[5.11.3]
in_predicate ::= value_expression [**not**] **in** subquery_or_value_spec_list

[5.6]
input_parameter ::= *Ada*_identifier_1 [named_phrase] :
[**in**] [**out**] domain_reference [**not null**]

[5.6]
input_parameter_list ::= (parameter { ; parameter })

[3.4]
input_reference ::= [procedure_reference .] *Ada*_identifier |
[cursor_proc_reference .] *Ada*_identifier

[5.8]
insert_column_list ::= insert_column_specification { , insert_column_specification }

[5.8]
insert_column_specification ::= column_name [named_phrase] [**not null**]

[5.9]
insert_from_clause ::= **from** into_from_body

[5.3]
insert_statement_query ::= **Insert into** table_name [(*SQL*_insert_column_list)]
query_specification

[5.3]
insert_statement_values ::= **Insert into** table_name [(insert_column_list)]
[insert_from_clause] **values** [(insert_value_list)]

[5.8]
insert_value ::= **null** |
constant_reference |
literal |
column_name |
domain_parameter_reference

[5.8]
insert_value_list ::= insert_value { , insert_value }

[5.9]
into_clause ::= **Into** ß

[5.9]
into_from_body ::= *Ada*_identifier_1 : record_id |
*Ada*_identifier_1 |
: record_id

[5.11.4]
like_predicate ::= column_reference [**not**] **like** pattern [escape_clause]

[3.1]
module ::= definitional_module | schema_module | abstract_module

[3.2]
module_name ::= *Ada*_identifier

[3.4]
module_reference ::= *Ada*_identifier

[4.1.5]
named_phrase ::= **named** *Ada*_identifier

[5.11.5]
null_predicate ::= column_reference **Is**[**not**] **null**

[5.5]
open_statement ::= **open** [*Ada*_identifier]

[4.1.1.3]
option ::= fundamental |
for word_list **use** pattern_list ; |
for word_list **use predefined** ;

[4.1.1.3]
options ::= { options }

[4.1.3]
parameter_association ::= *Ada*_identifier => static_expression |
map => database_mapping |
enumeration => enumeration_reference |
scale => static_expression |
length => static_expression

[4.1.3]
parameter_association_list ::= parameter_association { , parameter_association }

[4.1.1.2]
pattern ::= domain_pattern |
subdomain_pattern |
derived_domain_pattern

[4.1.1.2]
pattern_element ::= character_literal

[4.1.1.2]
pattern_list ::= pattern_element { pattern_element }

[5.11.4]
pattern_string ::= value_spec

[4.1.1.2]
patterns ::= { pattern }

[5.11]
predicate ::= comparison_predicate |
between_predicate |
in_predicate |
like_predicate |
null_predicate |
quantified_predicate |
exists_predicate

[5.10]
primary ::= literal |
constant_reference |
domain_parameter_reference |
column_reference |
input_reference |
set_function_specification |
domain_conversion |
(value_expression)

[5.2]
procedure_declaration ::= [**extended**]
procedure *Ada*_identifier_1
[input_parameter_list]
is
statement
[status_clause]
;

[5.1]
procedure_or_cursor ::= cursor_declaration | procedure_declaration

[3.4]
procedure_reference ::= [module_reference .] *Ada*_identifier

[5.11.6]
quantified_predicate ::= value_expression comp_op quantifier subquery

[5.11.6]
quantifier ::= **all** | **some** | **any**

[5.4]
query ::= query_expression | extended_query_expression

[5.4]
query_expression ::= query_term |
query_expression **union** [**all**] query_term

[4.2.2]
query_spec ::= query_specification | extended_query_specification

```
[ 5.4 ]
query_specification ::=  select [ distinct | all ] select_list
                             from_clause
                             [ where search_condition ]
                             [ SQL_group_by_clause ]
                             [ having search_condition ]

[ 5.4 ]
query_term ::=  query_specification |
                ( query_expression )

[ 4.1.5 ]
record_declaration ::=  record Ada_identifier_1 [ named_phrase ] is
                            component_declarations
                        end [ Ada_identifier_2 ] ;

[ 5.9 ]
record_id ::=  new Ada_identifier_2  |
               record_reference

[ 3.4 ]
record_reference ::= [ module_reference . ] Ada_identifier

[ 5.12 ]
result_expression ::= value_expression | *

[ 5.3 ]
rollback_statement ::= rollback work

[ 4.2 ]
schema_element ::=  table_definition          |
                    view_definition           |
                    SQL_privilege_definition  |
                    extended_schema_element

[ 4.2 ]
schema_module ::=  [ context ]
                   [ extended ]
                   schema module SQL_identifier_1 is
                       { schema_element }
                   end [ SQL_identifier_2 ] ;

[ 3.2 ]
schema_name ::= SQL_identifier

[ 3.3 ]
schema_ref ::= schema_name | Ada_identifier

[ 3.4 ]
schema_reference ::= schema_name | Ada_identifier

[ 5.11 ]
search_condition ::= boolean_term | search_condition or boolean_term

[ 5.7 ]
select_list ::= * | select_parameter { , select_parameter }
```

[5.7]
select_parameter ::= value_expression [named_phrase] [**not null**]
[**dblength** [named_phrase]]

[5.3]
select_statement ::= **select** [**distinct** | **all**] select_list
[into_clause]
from_clause
[**where** search_condition]
[*SQL*_group_by_clause]
[**having** search_condition]

[5.10]
set_function_specification ::= **count** (*) |
distinct_set_function |
all_set_function

[5.3]
set_item ::= column_reference = update_value

[4.2.1]
*SQL*_default_clause ::= *(see [SQL] 6.4)*

[5.3]
*SQL*_group_by_clause ::= **group by** column_reference { , column_reference }

[5.3]
*SQL*_insert_column_list ::= column_name { , column_name }

[5.4]
*SQL*_order_by_clause ::= **order by** *SQL*_sort_specification { , *SQL*_sort_specification }

[4.2]
*SQL*_privilege_definition ::= *(see [SQL] 6.10)*

[4.2.1]
*SQL*_reference_specification ::= *(see [SQL] 6.7)*

[4.2.1]
*SQL*_referential_constraint_definition ::= *(see [SQL] 6.7)*

[5.4]
*SQL*_sort_specification ::= Unsigned_integer_literal [**asc** | **desc**] |
column_reference [**asc** | **desc**]

[4.2.1]
*SQL*_unique_constraint_definition ::= *(see [SQL] 6.6)*

[4.2.1]
*SQL*_unique_specification ::= *(see [SQL] 6.6)*

[4.1.8]
sqlcode_assignment ::= static_expression_list => enumeration_literal |
static_expression_list => **raise** exception_reference

```
[ 5.2 ]
statement ::=   commit_statement          |
                delete_statement         |
                insert_statement_values  |
                insert_statement_query   |
                rollback_statement       |
                select_statement         |
                update_statement         |
                extended_statement

[ 4.1.4 ]
static_expression ::= value_expression

[ 4.1.8 ]
static_expression_list ::=   static_expression { , static_expression } |
                             static_expression .. static_expression

[ 5.13 ]
status_clause ::= status status_reference [ named_phrase ]

[ 4.1.8 ]
status_map_declaration ::= status Ada_identifier_1
                               [ named_phrase ]
                               [ uses target_enumeration ]
                           is ( sqlcode_assignment { , sqlcode_assignment ) ;

[ 3.4 ]
status_reference ::= [ module_reference . ] Ada_identifier

[ 4.1.3 ]
subdomain_declaration ::= subdomain Ada_identifier is dom_ref [ not null]
                              [ ( parameter_association_list ) ] ;

[ 4.1.1.2 ]
subdomain_pattern ::= subdomain pattern is pattern_list
                      end pattern ;

[ 3.4 ]
subdomain_reference ::= [ module_reference . ] Ada_identifier

[ 5.12 ]
subquery ::= ( select [ distinct | all ] result_expression
             from_clause
             [ where search_condition ]
             [ SQL_group_by_clause ]
             [ having search_condition ] )

[ 5.11.3 ]
subquery_or_value_spec_list ::= subquery | ( value_spec_list )

[ 4.2.1 ]
table_constraint_definition ::=   SQL_unique_constraint_definition       |
                                  SQL_referential_constraint_definition  |
                                  check_constraint_definition
```

[4.2.1]
table_definition ::= **[extended] table** *SQL*_identifier_1 **is**
table_element { , table_element }
end [*SQL*_identifier_2] ;

[4.2.1]
table_element ::= column_definition |
table_constraint_definition |
extended_table_element

[3.3]
table_name ::= [schema_ref .] *SQL*_identifier

[3.3]
table_ref ::= table_name [[**as**] correlation_name]

[3.4]
table_reference ::= [schema_reference .] *SQL*_identifier

[4.1.8]
target_enumeration ::= enumeration_reference | **boolean**

[5.10]
term ::= factor |
term * factor |
term / factor

[4.1.1.3]
type ::= **dbms** | **not null** | **null**

[5.3]
update_statement ::= **update** table_name
set set_item { , set_item }
[**where** search_condition]

[5.3]
update_value ::= **null** | value_expression

[3.2]
use_clause ::= **use** module_name { , module_name } ;

[5.11.1]
val_or_subquery ::= value_expression | subquery

[5.10]
value_expression ::= term |
value_expression + term |
value_expression - term

[5.11.3]
value_spec ::= input_reference |
static_expression |
user

[5.11.3]
value_spec_list ::= value_spec { , value_spec }

[4.2.2]
view_definition ::= [**extended**] **view** *SQL*_identifier_1 **as** query_spec
[**with check option**]
end [*SQL*_identifier_2] ;

[3.2]
with_clause ::= **with** module_name [as_phrase]
{ , module_name [as_phrase] } ;

[3.2]
with_schema_clause ::= **with schema** schema_name [as_phrase]
{ , schema_name [as_phrase] } ;

[4.1.1.3]
word_list ::= **context clause** |
null value |
null_bearing assign |
not_null_bearing assign

Index